Principles of Compiler Design

A. A. Puntambekar

Technical Publications Pune®

Principles of Compiler Design

First Edition : August 2006

ISBN 81 - 8431 - 064 - 1

Published by :

Technical Publications Pune

#1, Amit Residency, 412, Shaniwar Peth,
Pune - 411 030, India.
Phone : (020) 24495496 , Tele/Fax : (020) 24495497
Email : technical@vtubooks.com

Printer :

Alert DTPrinters
Omkar Compound,
Sr.no. 10/3,
Pune Sinhagad Road,
Pune - 411 041

Head office

Technical Publications
1, Amit Residency, 412,Shaniwar Peth, Pune - 411030, India
(020) 24495496, 66023931,Tele/Fax : (020) 24495497

Bangalore

Pragati Book House
1676-14th, Main Road, Prakash Nagar,
Bangalore - 560021, India
Phone : (080) 23324037, Fax : 23324848

Jalgaon

Pragati Books Pvt. Ltd.
34, V. V. Golani Market, Navi Peth,
Jalgaon - (02587) 425 001, India
Phone : 220395

Mumbai

Pragati Books Pvt. Ltd.
385, S.V.P. Road, Rasdhara,
Co-op. Society, Girgaum, Mumbai - 400 004
India. Phone : (022) 23869976, 23856339

Nagpur

Pratibha Book Distributors
Shop No. 3, 1st Floor Lokratna, Commercial Complex,
Zansi Rani Square, Sitabuldi,
Nagpur - 440012, India. Phone : (0712) 547129

Hyderabad

Nirali Book House
22,Shivam Enclave, 4-5 - 947,
Badi Chavadi, Hyderabad - 500095
India. Phone : (040) 5545313

Nasik

Pragati Books Pvt. Ltd.
741, Gaydhani Sankul, First Floor,
Raviwar Karanja, Nasik - 422001
India. Phone : (0253) 2506438

Chennai

Pragati Books
9/1, Montieth Road, Behind Taas Mahal,
Egmore, Chennai - 600 008,
India. Ph. - (044) 5518 3535 Mobile - 94440 01782

Pune

Pragati Books Pvt. Ltd.
119, Budhwar Peth,
Jogeshwari Mandir lane, Pune - 411 002, India
Phone : (020) 24452044

Technical Books Distributor
B - Ground floor, 'QUANTA ANNAM'
24th Street, H-965/966, Opp. U C MAS
Anna Nagar (W), Chennai - 600040, India.
Phone : (044) 26161903, Mobile No. : 9840324419

Pragati Impex (Export Division)
21, Amit Residency, 412, Shaniwar Peth,
Pune - 411030, India
+919845021552, +919370286000

This edition can be exported from India only.

Table of Contents

1 Introduction to Compiling

1.1 Introduction

Compilers are basically translators. Designing a compiler for some language is a complex and time consuming process. Since new tools for writing the compilers are available this process has now become a sophisticated activity. While studying the subject compiler it is necessary to understand what is compiler and how the process of compilation can be carried out. In this chapter we will get introduced with the basic concepts of compiler. We will see how the source program is compiled with the help of various phases of compiler. Lastly we will get introduced with various compiler construction tools.

1.2 Why to Write Compiler ?

Compiler is a program which takes one language (source program) as input and translates it into an equivalent another language (target program). During this process of translation if some errors are encountered then compiler displays them as error messages. The basic model of compiler can be represented as follows –

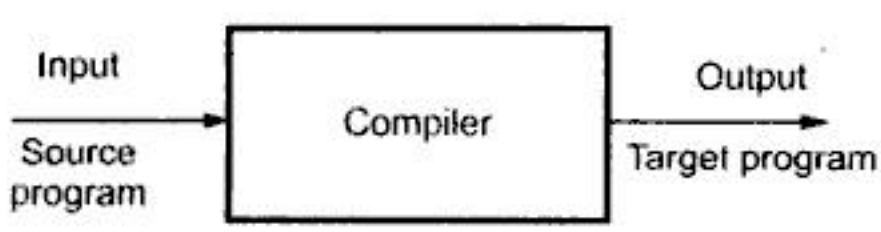


Fig. 1.1 Basic model of compiler

The compiler takes a source program as higher level languages such as C, PASCAL, FORTRAN and converts it into low level language or a machine level language such as assembly language.

1.2.1 Compiler: Analysis - Synthesis Model

The compilation can be done in to parts: analysis and synthesis. In analysis part the source program is read and broken down into constituent pieces. The meaning of the source string is determined and then an intermediate code is created from the input source program. In synthesis part this intermediate form of the source language is taken and converted into an equivalent target program. The analysis and synthesis model is as shown in fig. 1.2.

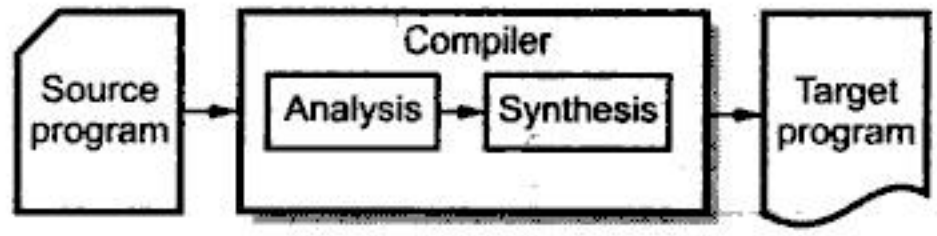


Fig. 1.2 Analysis - Synthesis model of compiler

The analysis part is carried out in three sub parts –

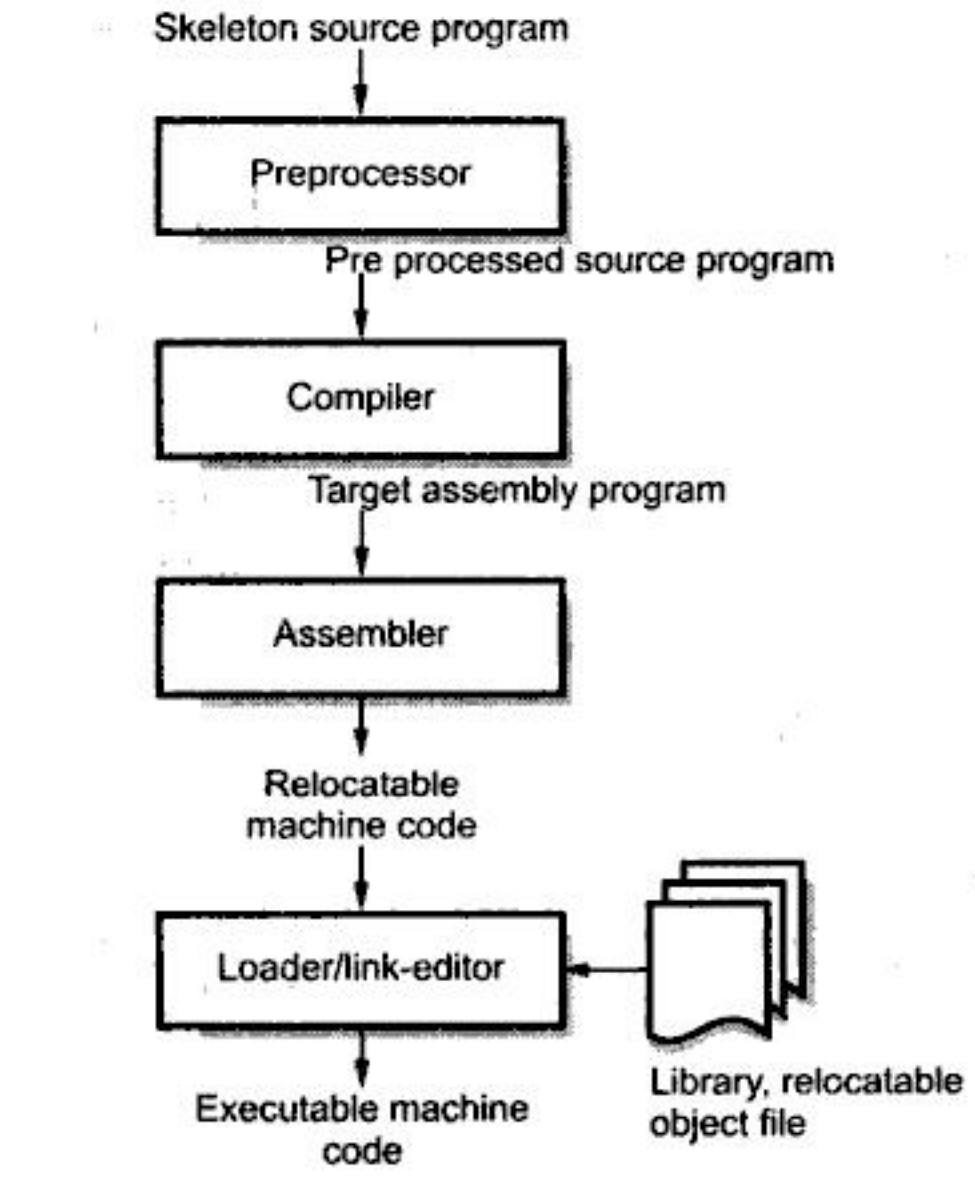
1. Lexical Analysis – In this step the source program is read and then it is broken into stream of strings. Such strings are called tokens. Hence tokens are nothing but the collection of characters having some meaning.
2. Syntax Analysis – In this step the tokens are arranged in hierarchical structure that ultimately helps in finding the syntax of the source string.
3. Semantic Analysis – In this step the meaning of the source string is determined.

In all these analysis steps the meaning of the every source string should be unique. Hence actions in lexical, syntax and semantic analysis are uniquely defined for a given language.

1.2.2 Execution of Program

To create an executable form of your source program only a compiler program is not sufficient. You may require several other programs to create an executable target program. The target program generated by the compiler is processed further before it can be run which is as shown in the Fig. 1.3 (see Fig. 1.3 on next page).

The compiler takes a source program written in high level language as an input and converts it into a target assembly language. The assembler then takes this assembly code as input and produces a relocatable machine code as output. Then a program called loader is called for performing the task of loading and link editing. The task of loader is to perform the relocation of an object code. Relocation of an object code means allocation of load time addresses which exist in the memory and placement of load time addresses and data in memory at proper locations. The link editor links the object modules and prepares a single module from several files of relocatable object modules to resolve the mutual references. These files may be library files and these library files may be referred by any program.



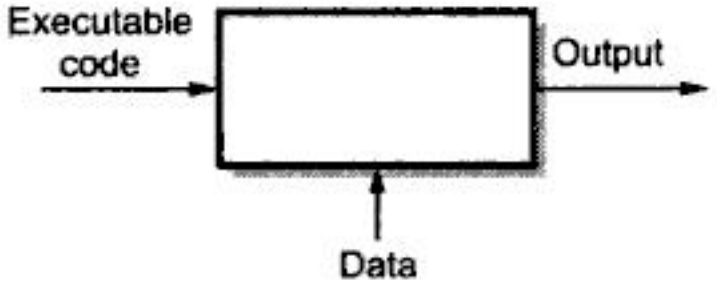


Fig. 1.3 Language processing Model

1.3 Compilation Process in Brief

1.3.1 The Phases of Compiler

As we have discussed earlier the process of compilation is carried out in two parts: analysis and synthesis. Again the analysis is carried out in three phases: Lexical analysis, syntax analysis and semantic analysis. And the synthesis is carried out with the help of intermediate code generation, code generation and code optimization. Let us discuss these phases one by one.

1. Lexical Analysis -

The lexical analysis is also called scanning. It is the phase of compilation in which the complete source code is scanned and your source program is broken up into group of strings called token. A token is a sequence of characters having a collective meaning. For example if some assignment statement in your source code is as follows,

total = count + rate *10

Then in lexical analysis phase this statement is broken up into series of tokens as follows-

1. The identifier total
2. The assignment symbol
3. The identifier count
4. The plus sign
5. The identifier rate
6. The multiplication sign
7. The constant number 10

The blank characters which are used in the programming statement are eliminated during the lexical analysis phase.

2. Syntax Analysis -

The syntax analysis is also called parsing. In this phase the tokens generated by the lexical analyser are grouped together to form a hierarchical structure. The syntax analysis determines the structure of the source string by grouping the tokens together. The hierarchical structure generated in this phase is called parse tree or syntax tree. For the expression total = count + rate *10 the parse tree can be generated as follows.

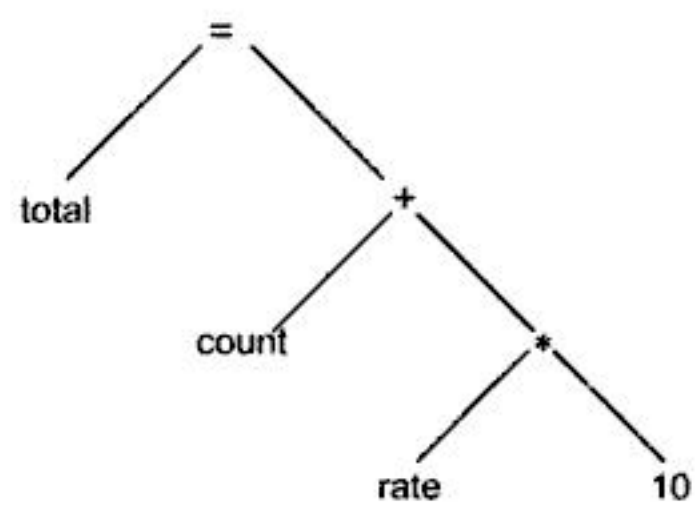


Fig. 1.4 Parse tree for total = count + rate * 10

In the statement 'total = count + rate *10' first of all rate*10 will be considered because in arithmetic expression the multiplication operation should be performed before the addition.And then the addition operation will be considered. For building such type of syntax tree the production rules are to be designed.The rules are usually expressed by context free grammar. For the above statement the production rules are –

(1) expression → identifier

(2) expression → number

(3) expression → expression1 + expression2

(4) expression → expression1* expression2

(5) expression → (expression1)

By rule (1) count and rate are expressions and by rule(2) 10 is also an expression. By rule (4) we get rate*10 as expression. And finally count +rate*10 is an expression.

3. Semantic Analysis

Once the syntax is checked in the syntax analyser phase the next phase i.e. the semantic analysis determines the meaning of the source string. For example meaning of source string means matching of parenthesis in the expression, or matching of if ...else statements or performing arithmetic operations of the expressions that are type compatible, or checking the scope of operation.

For example,

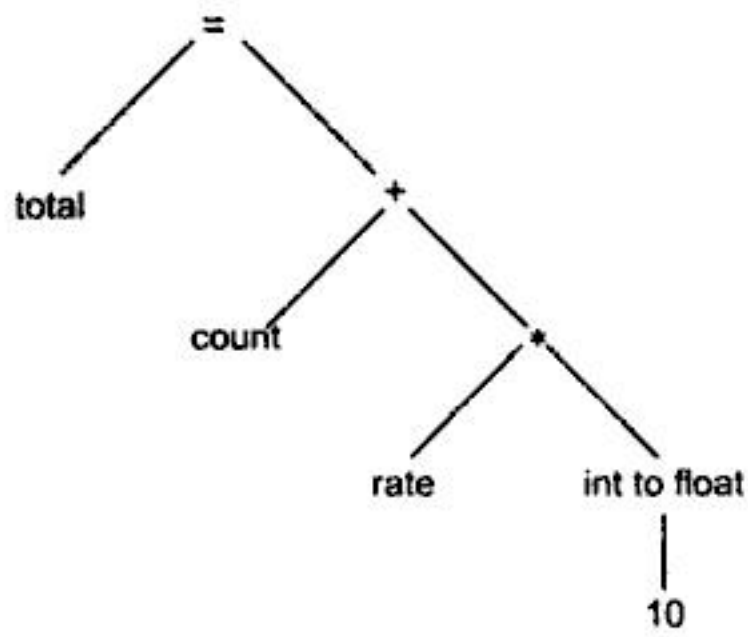


Fig. 1.5 Semantic analysis

Thus these three phases are performing the task of analysis. After these phases an intermediate code gets generated.

4. Intermediate code generation –

The intermediate code is a kind of code which is easy to generate and this code can be easily converted to target code. This code is in variety of forms such as three address code, quadruple, triple, posix. Here we will consider an intermediate code in three address code form. This is like an assembly language. The three address code consists of instructions each of which has at the most three operands. For example,

$$t_1 := \text{int to float (10)}$$

$$t_2 := \text{rate} * t_1$$

$$t_3 := \text{count} + t_2$$

$$\text{total} := t_3$$

There are certain properties which should be possessed by the three address code and those are,

a. Each three address instruction has at the most one operand in addition to the assignment. Thus the compiler has to decide the order of the operations devised by the three address code.

b. The compiler must generate a temporary name to hold the value computed by each instruction.

c. Some three address instructions may have fewer than three operands for example first and last instruction of above given three address code.

5. Code optimization -

The code optimization phase attempts to improve the intermediate code. This is necessary to have a faster executing code. Thus by optimizing the code the overall running time of the target program can be improved.

6. Code generation -

In code generation phase the target code gets generated. The intermediate code instructions are translated into sequence of machine instructions.

```
MOV     rate,  R1
MUL     # 10.0, R1
MOV     count, R2
ADD     R2, R1
MOV     R1, total.
```

Symbol table management

To support these phases of compiler a symbol table is maintained. The task of symbol table is to store identifiers (variables) used in the program. The symbol table also stores information about attributes of each identifier. The attributes of identifiers are usually its type, its scope, information about the storage allocated for it. The symbol table also stores information about the subroutines used in the program. In case of subroutine, the symbol table stores the name of the subroutine, number of arguments passed to it, type of these arguments, the method of passing these arguments(may be call by value or call by reference) return type if any. Basically symbol table is a data structure used to store the information about identifiers. The symbol table allows us to find the record for each identifier quickly and to store or retrieve data from that record efficiently.

During compilation the lexical analyzer detects the identifier and makes its entry in the symbol table. However, lexical analyzer can not determine all the attributes of an identifier and therefore the attributes are entered by remaining phases of compiler.

Various phases can use the symbol table in various ways. For example while doing the semantic analysis and intermediate code generation, we need to know what type of identifiers are. The during code generation typically information about how much storage is allocated to identifier is seen.

Error detection and handling

To err is human. As programs are written by human beings therefore they can not be free from errors. In compilation, each phase detects errors. These errors must be reported to error handler whose task is to handle the errors so that the compilation can proceed. Normally, the errors are reported in the form of message. When the input characters from the input do not form the token, the lexical analyzer detects it as error. Large number of errors can be detected in syntax analysis phase. Such errors are popularly called as syntax errors. During semantic analysis; type mismatch kind of error is usually detected.

Example of compilation

For statement a = b + c * 60; write all compilation phases with input and output to each phase.

Fig. 1.6 see on Next Page .

Front end back end

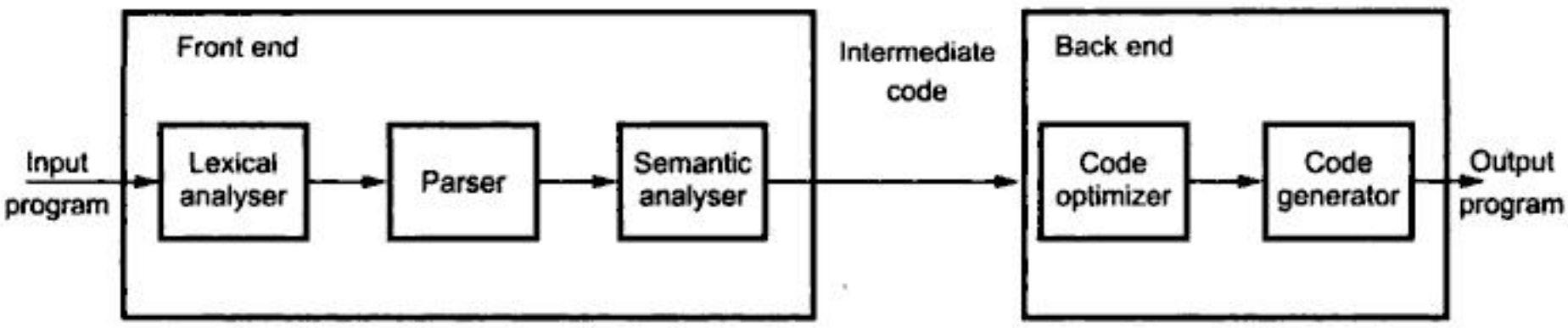


Fig. 1.6(a) Front end-Back end model

Different phases of compiler can be grouped together to form a front end and back end. The front end consists of those phases that primarily dependant on the source language and independent on the target language. The front end consists of analysis part. Typically it includes lexical analysis, syntax analysis, and semantic analysis. Some amount of code optimization can also be done at front end. The back end consists of those phases that are totally dependent upon the target language and independent on the source language. It includes code generation and code optimization. The front end back end model of the compiler is very much advantageous because of following reasons –

1. Keeping the same front end and attaching different back ends one can produce a compiler for same source language on different machines.

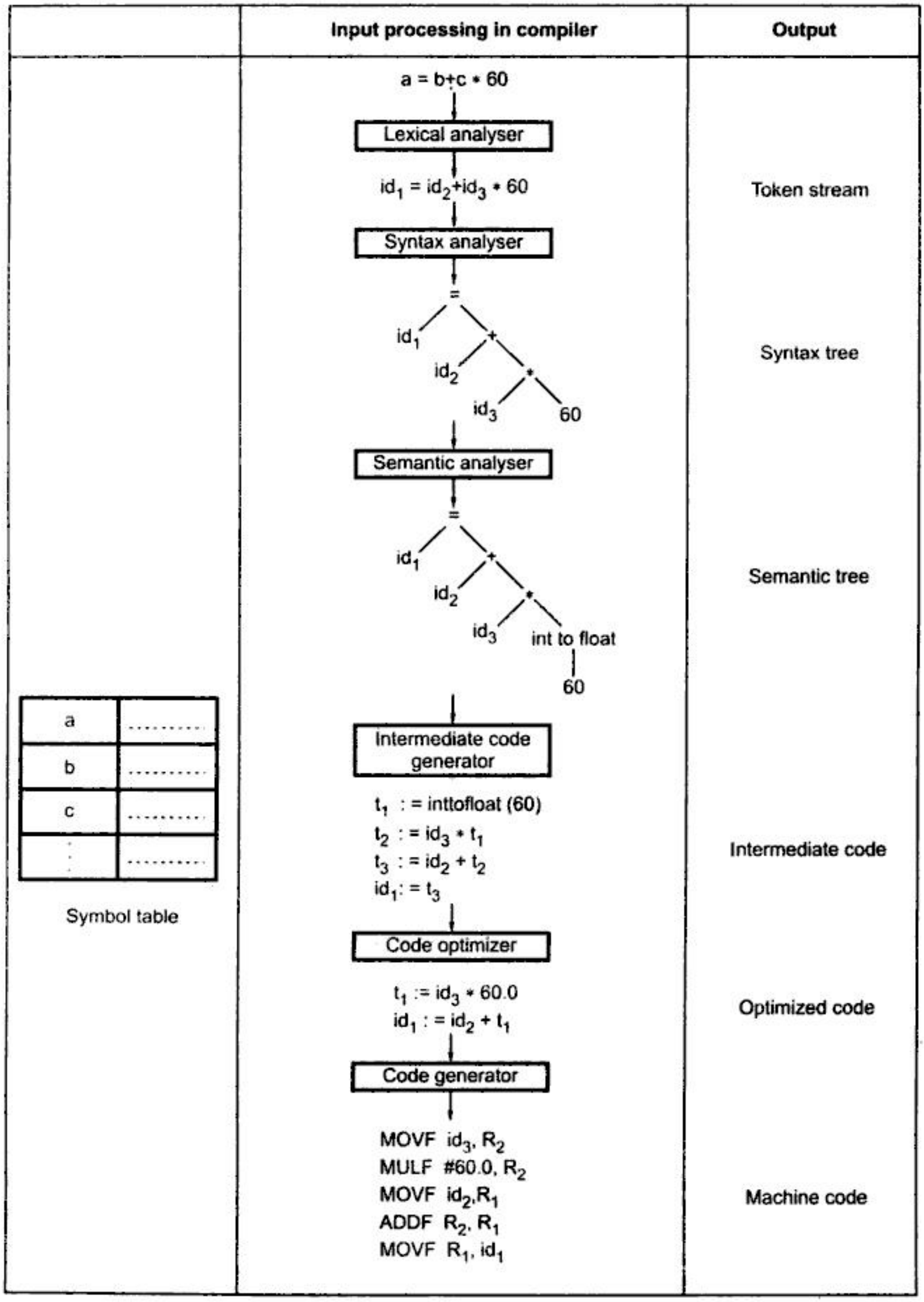


Fig. 1.6

2. Keeping different front ends and same back end one can compile several different languages on the same machine.

1.4 Cousins of the Compiler

Sometimes the output of preprocessor may be given as input to the compiler. Cousins of compiler means the context in which the compiler typically operates. Such contexts are basically the programs such as preprocessor, assembiers, loaders and link editors. Let us discuss them in detail.

1. Preprocessors – The output of preprocessors may be given as the input to compilers. The tasks performed by the preprocessors are given as below-

- Preprocessors allow user to use macros in the program. Macro means some set of instructions which can be used repeatedly in the program. Thus macro preprocessing task is be done by preprocessors.
- Preprocessor also allows user to include the header files which may be required by the program.

  For example #include<stdio.h>

  By this statement the header file stdio.h can be included and user can make use of the functions defined in this header file. This task of preprocessor is called file inclusion.

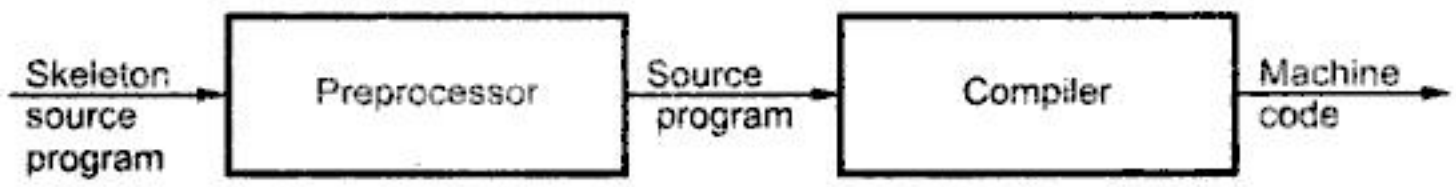


Fig. 1.7 Role of preprocessor

For a macro there are two kinds of statements macro definition and macro use. The macro definition is given by keyword like "define" or "macro" followed by the name of the macro.

For example :

define P 3.14

2. Assemblers – Some compilers produce the assembly code as output which is given to the assemblers as an input. The assembler is a kind of translator which takes the assembly program as input and produces the machine code as output.

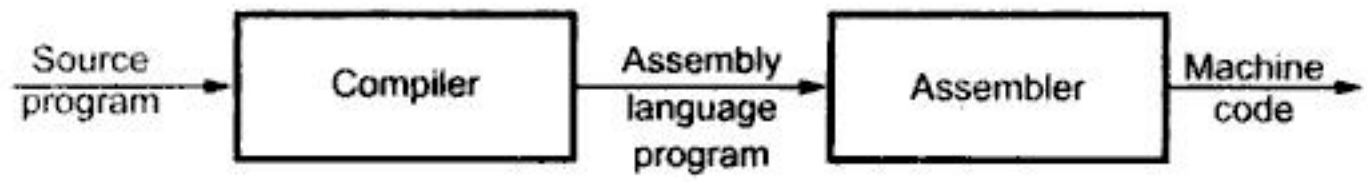


Fig. 1.8 Role of assembler

An assembly code is a mnemonic version of machine code. The typical assembly instructions are as given below –

```
MOV  a, R1
MUL  #5,R1
ADD  #7,R1
MOV  R1,b
```

The assembler converts these instructions in the binary language which can be understood by the machine. Such a binary code is often called as machine code. This machine code is a relocatable machine code that can be passed directly to the loader/linker for execution purpose.

The assembler converts the assembly program to low level machine language using two passes. A pass means one complete scan of the input program. The end of second pass is the relocatable machine code.

3. Loaders and Link Editors – Loader is a program which performs two functions: loading and link editing. Loading is a process in which the relocatable machine code is read and the relocatable addresses are altered. Then that code with altered instructions and data is placed in the memory at proper location. The job of link editor is to make a single program from several files of relocatable machine code. If code in one file refers the location in another file then such a reference is called external reference. The link editor resolves such external references also.

1.5 Concept of Pass

One complete scan of the source language is called pass. It includes reading an input file and writing to an output file. Many phases can be grouped one pass. It is difficult to compile the source program into a single pass, because the program may have some forward references. It is desirable to have relatively few passes, because it takes time to read and write intermediate file. On the other hand if we group several phases into one pass we may be forced to keep the entire program in the memory. Therefore memory requirement may be large.

In the first pass the source program is scanned completely and output will be an easy to use form which is equivalent to the source program along with the additional

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

1.9 Comparison of Compilers and Interpreters

In interpreter the analysis phase is same as compiler i.e. lexical, syntactic and semantic analysis is also performed in interpreter. In compiler the object code needs to be generated whereas in interpreter the object code needs not to be generated. During the process of interpretation the interpreter exists along with the source program. Binding and type checking is dynamic in case of interpreter.

The interpretation is less efficient than compilation. This is because the source program has to be interpreted every time it is to be executed and every time it requires analysis of source program. The interpreter can be made portal as they do not produce the object code which is not possible in case of compiler. Interpreters save time in assembling and linking. Self modifying program can be interpreted but can not be compiled. Design of interpreter is simpler than the compiler. Interpreters give us improved debugging environment as it can check the errors like out of bound array indexing at run time. Interpreters are most useful in the environment where dynamic program modification is required.

1.10 Compiler Construction Tools

Writing a compiler is tedious and time consuming task. There are some specialized tools for helping in implementation of various phases of compilers. These tools are called compiler construction tools. These tools are also called as compiler-compiler, compiler-generators, or translator writing system. Various compiler construction tools are given as below-

1. Scanner generator – These generators generate lexical analysers. The specification given to these generators are in the form of regular expressions.

   The UNIX has utility for a scanner generator called LEX. The specification given to the LEX consists of regular expressions for representing various tokens.

2. Parser generators – These produce the syntax analyzer. The specification given to these generators is given in the form of context free grammar. Typically UNIX has a tool called YACC which is a parser generator.
3. Syntax-directed translation engines - In this tool the parse tree is scanned completely to generate an intermediate code. The translation is done for each node of the tree.
4. Automatic code generator – These generators take an intermediate code as input and converts each rule of intermediate language into equivalent machine language. The template matching technique is used. The intermediate code statements are replaced templates that represent the corresponding sequence of machine instructions.

5. Data flow engines – The data flow analysis is required to perform good code optimization. The data flow engines are basically useful in code optimization.

In this chapter we have discussed the fundamental concept of compiler. Now we will discuss every phase of compiler in detail in further chapters.

Review Questions

1. *What is compiler?*
2. *Explain analysis -synthesis model of compiler?*
3. *With a neat block diagram, explain various phases of compiler.*
4. *What are the advantages of front-end and back end of compiler ?*
5. *Explain the concept of compiler-compiler.*

2 Lexical Analysis

2.1 Introduction

The process of compilation starts with the first phase called lexical analysis. In this phase the input is scanned completely in order to identify the tokens. The token structure can be recognized with the help of some diagrams. These diagrams are popularly known as finite automata. And to construct such finite automata regular expressions are used. These diagrams can be translated into a program for identifying tokens. In this chapter we will see what the role of lexical analyzer in compilation process is. We will also discuss the method of identifying tokens from source program. Finally we will learn a tool, LEX which automates the construction of lexical analyzer.

2.2 Role of Lexical Analyzer

Lexical analyzer is the first phase of compiler. The lexical analyzer reads the input source program from left to right one character at a time and generates the sequence of tokens. Each token is a single logical cohesive unit such as identifier, keywords, operators and punctuation marks. Then the parser to determine the syntax of the source program can use these tokens. The role of lexical analyzer in the process of compilation is as shown below –

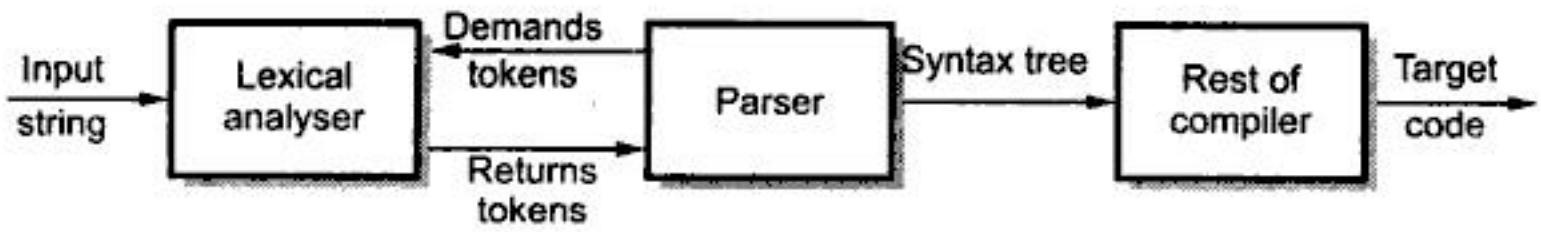


Fig. 2.1 Role of lexical analyzer

As the lexical analyzer scans the source program to recognize the tokens it is also called as scanner. Apart from token identification lexical analyzer also performs following functions.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- L^* is a set of strings having all the strings including ε.
- L^+ is a set of strings except ε.

2.4.3 Regular Expressions

Regular are mathematical symbolisms which describe the set of strings of specific language. It provides convenient and useful notation for representing tokens. Here are some rules that describe define the regular expressions over the input set denoted by Σ.

1. ε is a regular expression that denotes the set containing empty string.
2. If R1 and R2 is regular expression then R = R1 + R2 (same can also be represented as R = R1 | R2) is also regular expression which represents union operation.
3. If R1 and R2 is regular expression the R = R1.R2 is also a regular expression which represents concatenation operation.
4. If R1 is a regular expression then R = R1* is also a regular expression which represents kleen closure.

A language denoted by regular expressions is said to be a regular set or a regular language. Let us see some examples of regular expressions.

➠ **Example 2.1 :** *Write a Regular Expression (R.E.) for a language containing the strings of length two over* $\Sigma = \{0, 1\}$.

Solution : R.E. = (0+1) (0+1)

➠ **Example 2.2 :** *Write a regular expression for a language containing stings which end with "abb" over ? = {a, b}*

Solution : R.E. = (a+b)*abb

➠ **Example 2.3 :** *Write a regular expression for a recognizing identifier.]*

Solution : For denoting identifier we will consider a set of letters and digits because identifier is a combination of letters or letter and digits but having first character as letter always. Hence R.E. can be denoted as,

R.E = letter (letter+digit)*

Where letter = (A, B,...Z, a, b,...z) and digit = (0,1,2,...9).

2.4.4 Notations used for Representing Regular Expressions

Regular expressions are tiny units, which are useful for representing the set of strings belonging to some specific language. Let us see notations used for writing the regular expressions.

1. One or more instances – To represent one or more instances + sign is used. If r is a regular expression then r^+ denotes one or more occurrences of r. For example set of strings in which there are one or more occurrences of 'a' over the input set {a} then the regular expression can be written as a^+. It basically denotes the set of {a,aa,aaa,aaaa,...}.
2. Zero or more instances – To represent zero or more instances * sign is used. If r is a regular expression then r* denotes zero or more occurrences of r. For example, set of strings in which there are zero or more occurrences of 'a' over the input set {a} then the regular expression can be written as a*. It basically denotes the set of {ε, a, aa, aaa,...}.
3. Character classes – A class of symbols can be denoted by [ ]. For example [012] means 0 or 1 or 2. Similarly a complete class of a small letters from a to z can be represented by a regular expression [a-z]. The hyphen indicates the range. We can also write a regular expression for representing any word of small letters as [a-z]*.

2.4.5 Non Regular Language

A language which can not be described by a regular expression is called a non regular language and the set denoted by such language is called non regular set.

There are some languages which can not be described by regular expressions. For example we cannot write a regular expression to check whether the given language is palindrome or not. Similarly we cannot write a regular expression to check whether the string is having balanced parenthesis.

Thus regular expressions can be used to denote only fixed number of repetitions. Unspecific information cannot be represented by regular expression.

2.5 Recognition of Tokens

For a programming language there are various types of tokens such as identifier, keywords, constants, and operators and so on. The token is usually represented by a pair token type and token value.

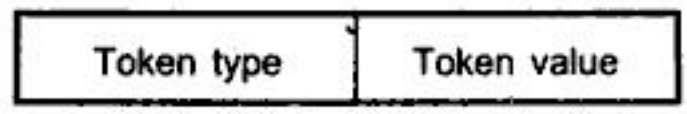


Fig. 2.7 Token representation

The token type tells us the category of token and token value gives us the information regarding token. The token value is also called token attribute. During lexical analysis process the symbol table is maintained. The token value can be a pointer to symbol table in case of identifier and constants. The lexical analyzer reads the input program and generates a symbol table for tokens.

For example –

We will consider some encoding of tokens as follows.

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to symbol table
constant	6	Ptr to symbol table
<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
–	9	2
=	10	–

Consider, a program code as

```
if(a<10)
     i=i+2;
else
     i=i-2;
```

Our lexical analyzer will generate following token stream.

1,(8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2,(5,107), 10, (5,107), (9,2), (6,110).

The corresponding symbol table for identifiers and constants will be,

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
{
printf("Usage:%s radius\n", argv[0]);
exit(1);
}
else
{
double radius = atof(argv[1]);
double area = area_of_circle(radius);
printf("Area of circle with radius %f = %f\n",radius,area);
}
return 0;//Returning 0
}
```

The program Area.C can be taken as input test program. This program can be scanned by generated lex.yy.C (We have now generated lex.yy.C using LEX tool). Hence the output obtained using following commands.

```
[root@localhost]# lex lexprog.l
[root@localhost]# cc lex.yy.c
[root@localhost]# ./a.Out Area.c
```

2.8 LEX Specification and Features

To generate the lexical analyzer the automation tool like LEX is the efficient way. The specification file for LEX is based on the pattern-action structure as shown below :

```
R1   {action1}
R2   {action2}
.
.
.
Rn   {actionn}
```

Where each R_i is a regular expression and each action $_i$ is a program fragment describing what action is to be taken for corresponding regular expression. Each action is a piece of program code that is to be executed whenever lexeme is matched by R_i .

Regular expression	Meaning
*	Matches with zero or more occurrences of preceeding expression. For example 1* occurrence of 1 for any number of times.
	Matches any single character other than new line character.
[ ]	A character class which matches any character within the bracket. For example [a-z] maches with any alphabet in lower case.
()	Group of regular expressions together put into a new regular expression.
" "	The string written in quotes matches literally. For example "Hanumaan" matches with the string Hanumaan.
$	Matches with the end of line as last character.

+	Matches with one or more occurrences of preceeding expression. For example [0-9]+ any number but not empty string.
?	Matches zero or one occurrence of preceding regular expression. For example [+-]?[0-9]+ a number with unary operator.
^	Matches the beginning of a line as first character.
[^]	Used as for negation. For example [^verb] means except verb match with anything else.
\	Used as escape metacharacter. For example \n is a newline character. \# prints the # literally
|	To represent the or i.e. another alternative. For example a | b means match with either a or b.

We have to construct recognizer that looks for the lexemes stored in the input buffer. It works using these two rules –

1. If more than one pattern matches then recognizer has to choose the longest lexeme matched.
2. If there are two or more patterns that match the longest lexeme, the first listed matching pattern is chosen.

Lexical analysis is essentially a process of recognizing different tokens from the source program. This process of recognition can be accomplished by building a classical model called Finite State Machine (FSM) or a Finite Automaton (FA). The characteristics of token can be represented by certain mathematical notation called "regular expressions". We have already discussed how to describe the strings belonging to certain language using regular expressions. We will see shortly that the finite state machines can be built for the regular expressions describing certain languages. The design of lexical analyzer involves three tasks-

(i) Representation of tokens by regular expression.

(ii) Representation of regular expressions by finite state machines.

(iii) Simulation of the behavior of finite state machine.

These three steps lead to the design of lexical analyzer. The LEX compiler constructs a transition table for a finite state machine from regular expression patterns in the LEX specification. The lexical analyzer itself consists of finite automaton simulator that uses the transition table. This transition table basically looks for the regular expression patterns from the input string stored in the input buffer. The finite automata can be deterministic or non-deterministic in nature.

A deterministic automaton is said to be deterministic if for each state S for every input $a \in \Sigma$ at the most one edge is labeled a.

For example :

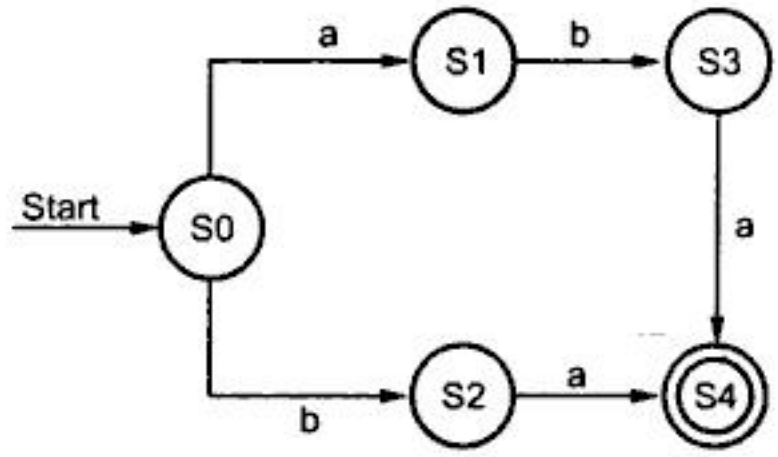


Fig. 2.18 DFA

A non deterministic finite automaton (NFA) is a set of states including start state and few or more final state and for a single input there can be multiple transitions. For example:

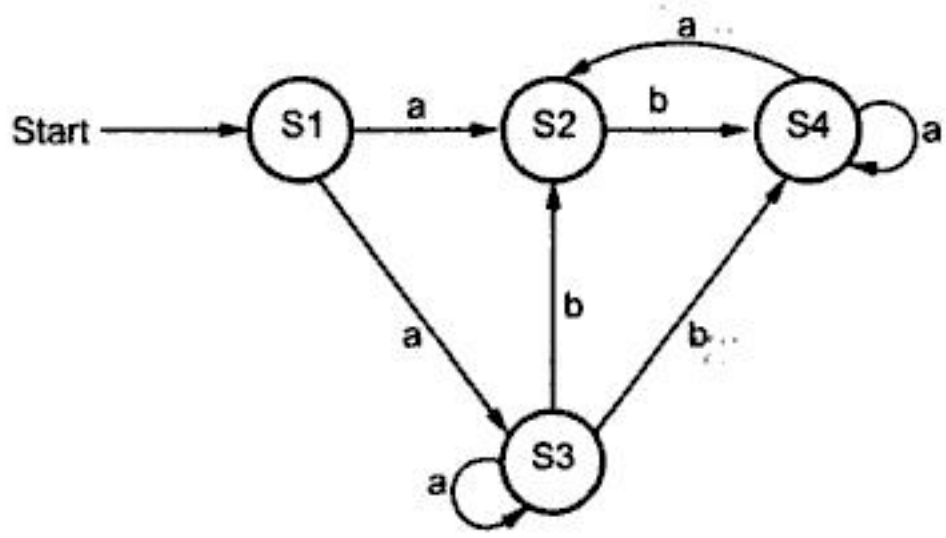


Fig. 2.19 NFA

To design the lexical analyzer generator first design the patterns of regular expressions for the tokens and then from these patterns it is easy to design a non-deterministic finite automata. But it is always easy to simulate the behavior of a DFA with a program; hence we have to convert the NFA drawn from the patterns to DFA.

The design of lexical analyzer generator is based on two approaches –

- Pattern matching using NFA.
- Using DFA for lexical analyzer.

Let us discuss these approaches one by one.

2.8.1 Pattern Matching based on NFA's

In this method the transition table for each NFA can be drawn and each NFA is designed based on the pattern mentioned in the specification file of LEX. Hence there can be n number of NFA's for n number of patterns. A start state is taken and with ε transitions all these NFAs can be connected together as shown below :

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Pattern 1 :

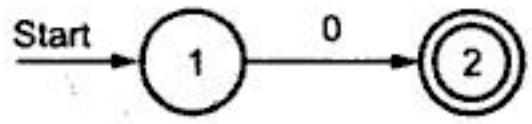


Fig. 2.21 (a)

Pattern 2 :

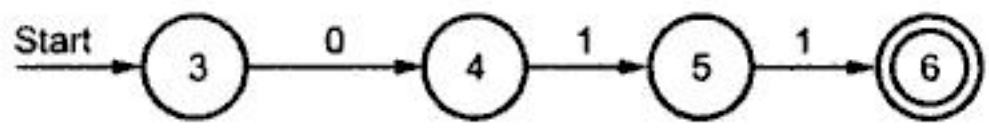


Fig. 2. 21 (b)

Pattern 3 :

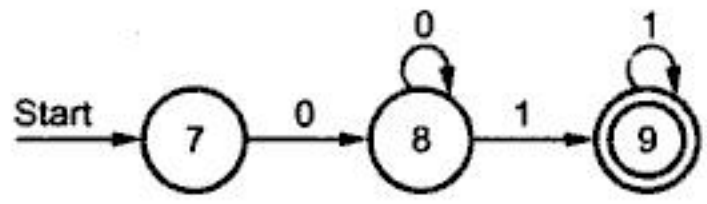


Fig. 2.21 (c)

The NFA can be drawn as a combination of pattern 1, pattern 2 and pattern 3 transition diagrams.

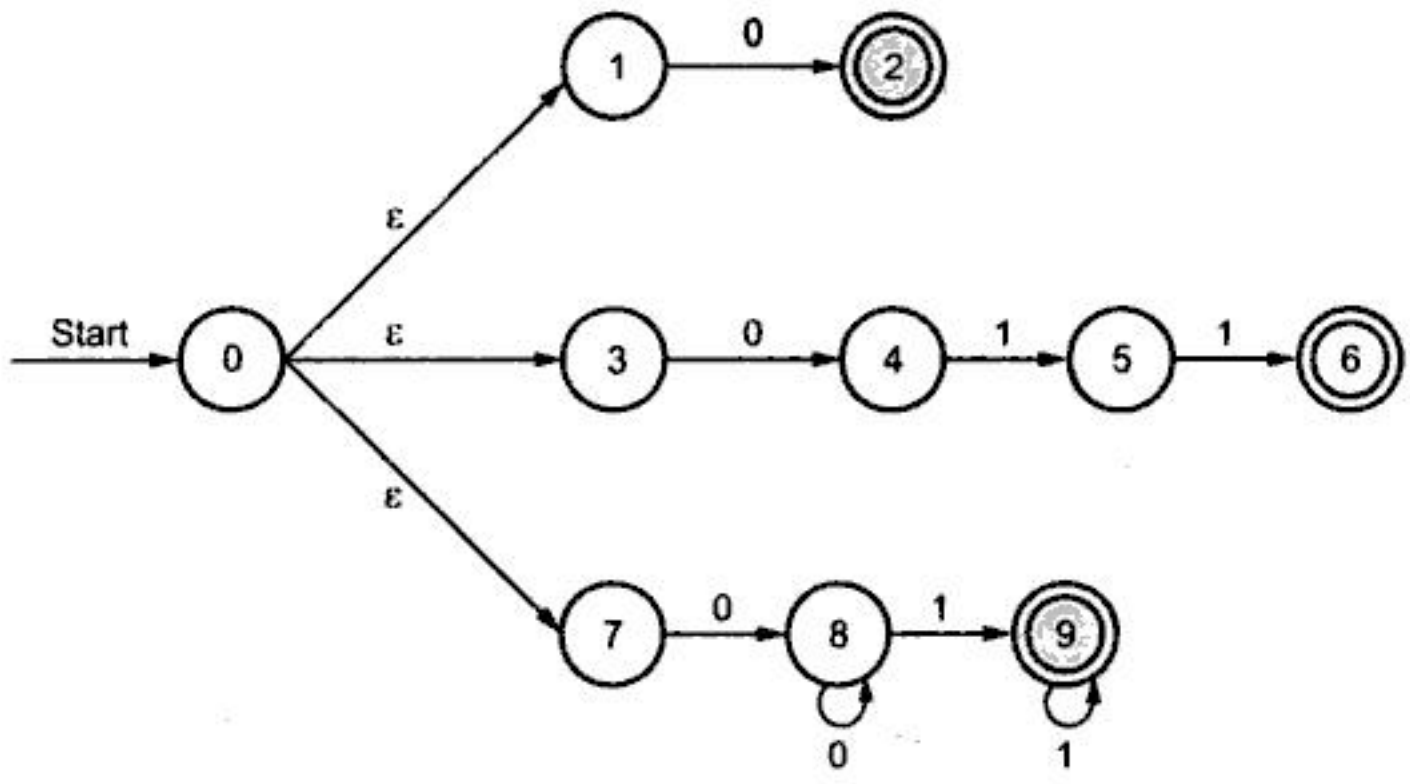


Fig. 2.21 (d) NFA with patterns for matching of input string

To summarize this one can say that LEX tries to match the input string with help of patterns one by one and it tries to match the longest lexeme.

2.8.2 DFA for Lexical Analyzers

In this method the DFA is constructed from the NFAs drawn from patterns of regular expressions. Then we try to match the input string with each of the DFA. The accepting states are recorded and we go on moving on the transitions until the input string terminates. If there are two patterns matching with the input string then the pattern which is mentioned first in the LEX specification file is always considered. For example consider the LEX specification file consists of

0 { }

011 { }

0^+1^+ { }

Now we have already built a NFA in earlier discussion for the above pattern. Let us design the DFA from NFA.

State	Input		Pattern matched
	0	1	
0137	248	–	–
248	8	59	0
8	8	9	–
59	–	69	0^+1^+
9	–	9	0^+1^+
69	–	9	011

For constructing DFA from the given NFA we begin with all the ε reachable states from state 0. These ε reachable states are grouped together to form a state 0137. Then we obtain the a and b transition for state 0137 then we obtain state 248 and NIL respectively. Thus we proceed with newly generated states. And the complete DFA can be designed.

To match the string 011 we get two matched pattern and those are 011 and 0^+1^+.

But the pattern 011 will be considered as the matched pattern because it appears before 0^+1^+ in the translation rules of the specification file.

As we have seen in the input buffer is used to store the input string. To recognize the lexeme actually two pointers are used: beginning pointer and forward pointer. The forward pointer moves in search of end of the current string (that can be white spaces or tabs or newline characters or some delimiters). This forward pointer when reaches to such termination mark it retracts and return the string between beginning pointer and forward pointer as token value. Thus the forward pointer is very much useful for

identifying the token. This pointer is also called as lookahead pointer. In the lexical analysis the lookahead operators are used in order to search the token from the input strings.

Review Questions

1. *What are the tasks of lexical analyzer?*
2. *Write a short note on : input buffering.*
3. *What is the role of regular expressions in lexical analyzer.*
4. *Give the block schematic of lexical analyzer?*
5. *Explain, how lexical analyzer is generated using LEX.*
6. *Write a LEX program to count number of characters, number of words and total number of lines from input text.*
7. *Write a LEX program to reverse the given input string.*

❑❑❑

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

3.1.4 Types of Grammar

The context free grammar G is a collection of following things

1. V is a set of non terminals
2. T is a set of terminals
3. S is a start symbol
4. P is a set of production rules.

Thus G can be represented as G=(V,T,S,P)

The production rules are given in following form-

Non Terminal $\rightarrow$ (V U T)*

➠ **Example 3.1 :** *Let the language* $L = a^n b^n$ *where* $n \geq 1$

Let G = (V,T,S,P)

Where V={S},

T= {a,b}

And S is a start symbol then

P =

S → aSb

S → ab

The production rules actually defines the language $a^n b^n$

The non terminal symbol occurs at the left hand side . These are the symbols which need to be expanded.The terminal symbols are nothing but the tokens used in the language. Thus any language construct can be defined by the context free grammar. For example if we want to define the declarative sentence using context free grammar then it could be as follows-

State → Type List Terminator

Type → int | float

List → List,id

List → id

Terminator → ;

Using above rules we can derive,

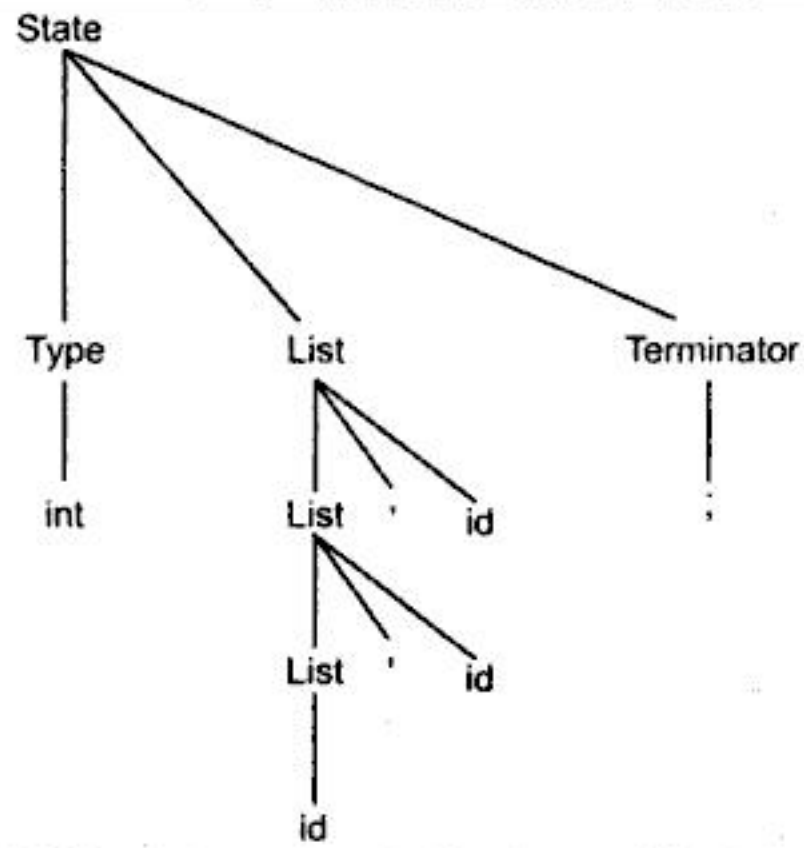


Fig. 3.4 Parse tree for derivation of int id, id, id;

Hence int id, id, id; can be defined by means of above Context Free Grammar.

Following rules to be followed while writing a CFG-

1) A single Non terminal should be at LHS.
2) The rule should be always in the form of LHS → RHS where RHS may be the combination of non terminal and terminal symbols.
3) The NULL derivation can be specified as NT → ε.
4) One of the non terminals should be start symbol and conventionally we should write the rules for this non terminal.

3.2 Concept of Derivation

Derivation from S means generation of string w from S. For constructing derivation two things are important.

i) Choice of Non terminal from several others.

ii) Choice of rule from production rules for corresponding non terminal.

Instead of choosing the arbitrary Non terminal one can choose either

i) Either leftmost Non terminal in a sentential form then it is called leftmost derivation.

ii) Or rightmost Non terminal in a sentential form, then it is called rightmost derivation.

For example:

Let G be a context free grammar for which the production rules are given as below-

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$

We have to derive the string aaabbabbba using above grammar.

Leftmost Derivation

$S \rightarrow aB$

$\rightarrow aaBB$

$\rightarrow aaaBBB$

$\rightarrow aaabSBB$

$\rightarrow aaabbABB$

$\rightarrow aaabbaBB$

$\rightarrow aaabbabB$

$\rightarrow aaabbabbS$

$\rightarrow aaabbabbbA$

$\rightarrow aaabbabbba$

Rightmost Derivation

$S \rightarrow aB$

$\rightarrow aaBB$

$\rightarrow aaBbS$

$\rightarrow aaBbbA$

$\rightarrow aaBbba$

$\rightarrow aaaBBba$

$\rightarrow aaaBbbba$

$\rightarrow aaabSbbba$

$\rightarrow aaabbAbbba$

$\rightarrow aaabbabbba$

The Parse Tree for the above derivations is as given below

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We can eliminate left recursion as

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid \varepsilon$

The grammar for arithmatic expression can be equivalently written as –

$E \rightarrow E + T \mid T$ $\quad\quad$ $E \rightarrow TE'$

$E' \rightarrow + TE' \mid \varepsilon$

$T \rightarrow FT'$

$T \rightarrow T * F \mid F$ $\quad\Rightarrow\quad$ $T' \rightarrow * FT' \mid \varepsilon$

$F \rightarrow CE$ $\quad\Rightarrow\quad$ $F \rightarrow (E) \mid id$

3) Left factoring

If the grammar is left factored then it becomes suitable for the use. Basically left factoring is used when it is not clear that which of the two alternatives is used to expand the non terminal. By left factoring we may be able to re-write the production in which the decision can be deferred until enough of the input is seen to make the right choice.

In general if

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

is a production then it is not possible for us to take a decision whether to choose first rule or second. In such a situation the above grammar can be left factored as

$A \rightarrow \alpha A'$

$A \rightarrow \beta_1 | \beta_2$

For example consider the following grammar -

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

The left factored grammar becomes

$S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \varepsilon$

$E \rightarrow b$

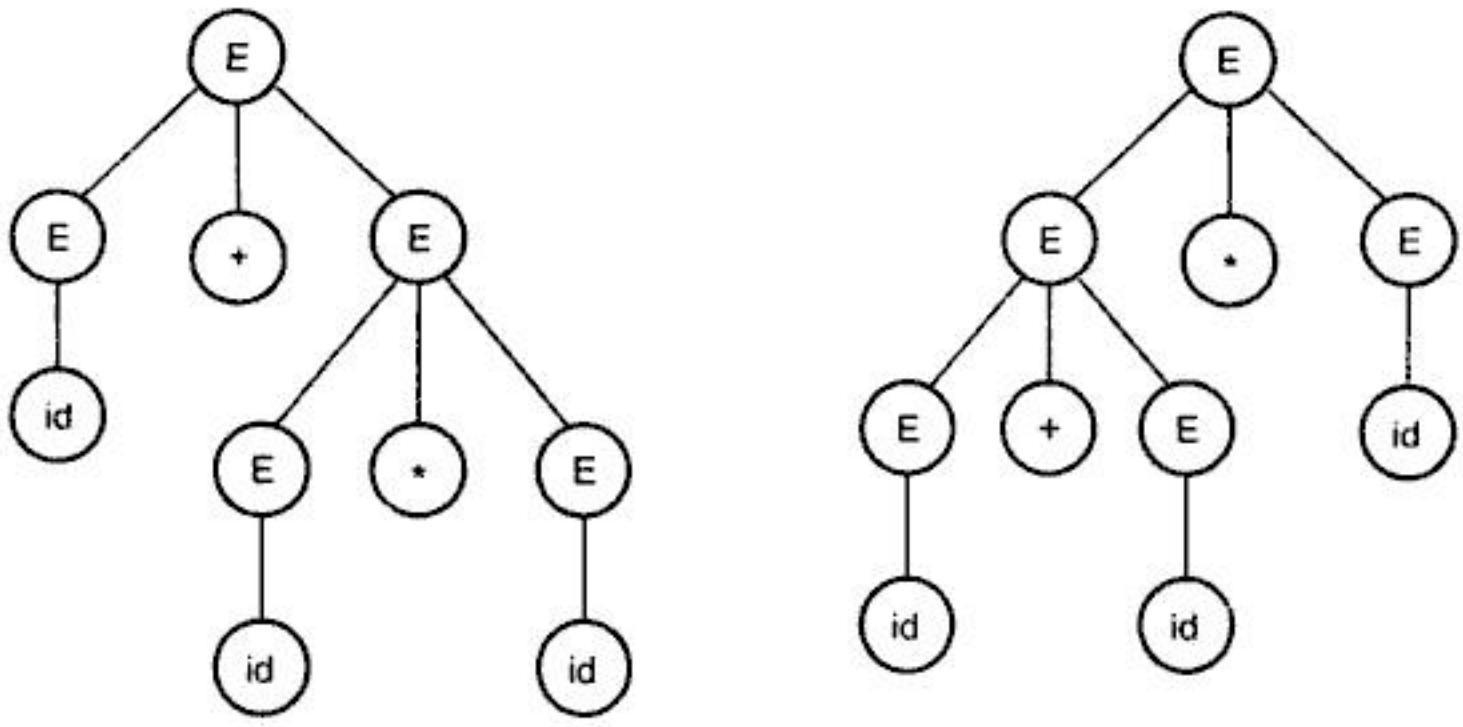
4) Ambiguity

The ambiguous grammar is not desirable in top down parsing. Hence we need to remove the ambiguity from the grammar if it is present.

For example :

$E \rightarrow E + \mid E * E \mid id$

is an ambiguous grammar. We will design the parse tree for id+id*id as follows



(a) Parse tree 1 (b) Parse tree 2

Fig. 3.12 Ambiguous grammer

For removing the ambiguity we will apply one rule :if the grammar has left associative operator(such as +, -,*, /) then induce the left recursion and if the grammar has right associative operator (exponential operator) then induce the right recursion.

The unambiguous grammar is

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$

Note one thing that the grammar is unambiguous but it is left recursive and elimination of such left recursion is again a must.

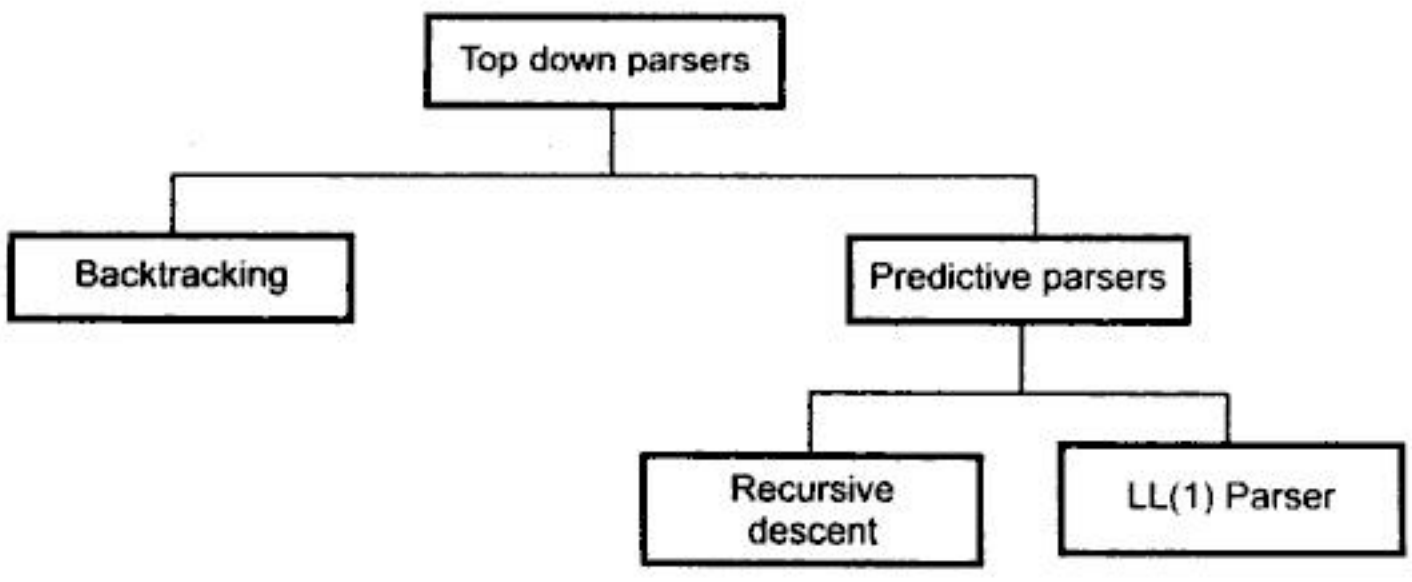


Fig. 3.13 Types Top-down parsers

There are two types by which the top down parsing can be performed

1. Backtracking
2. Predictive Parsing

A backtracking parser will try different production rules to find the match for the input string by back tracking each time. The backtracking is powerful than predictive parsing. But this technique is slower and it requires exponential time in general. Hence Backtracking is not preferred for practical compilers.

As the name suggests the predictive parser tries to predict the next construction using one or more lookahead symbols from input string. There are two types of Predictive parsers :

1. Recursive Descent
2. LL(1) Parser.

Let us discuss these types along with some example

3.5.2 Recursive Descent Parser

A parser that uses collection of recursive procedures for parsing the given input string is called Recursive Descent (RD) Parser. In this type of parser the CFG is used to build the recursive routines. The RHS of the production rule is directly converted to a program. For each non terminal a separate procedure is written and body of the procedure (code) is RHS of the corresponding non terminal.

Basic Steps for construction of RD parser

The RHS of the rule is directly converted into program code symbol by symbol.

1. If the input symbol is non terminal then a call to the procedure corresponding to the non terminal is made.
2. If the input symbol is terminal then it is matched with the lookahead from input. The lookahead pointer has to be advanced on matching of the input symbol.
3. If the production rule has many alternates then all these alternates has to be combined into a single body of procedure.
4. The Parser should be activated by a procedure corresponding to the start symbol.

Let us take one example to understand the construction of RD parser.Consider the grammar having start symbol E,

$$E \rightarrow num\ T$$

$$T \rightarrow *\ num\ T \mid \varepsilon$$

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

FIRST(FT')=FIRST(F)={(,id}

Hence M[F,(]= T→FT'

And M[F,id]= T→FT'

T → *FT'

A →α

A = T, α = *FT'

FIRST(*FT')={*}

Hence M[T,*] = T → *FT'

T'→ε

A →α

A=T', α =ε

FOLLOW(T')={+,),$}

Hence M[T',+] = T'→ε

M[T',)] = T'→ε

M[T',$] = T'→ε

F →(E)

A →α

A=F, α=(E)

FIRST((E))={ (}

Hence M[F,(]= F→(E)

F → id

A →α

A = F, α = id

FIRST(id)={ id }

Hence M[F,id]= F → id

The complete table can be as shown below –

	id	+	*	(	)	$
E	E → TE'	Error	Error	E → TE'	Error	Error
E'		E → +TE'	Error	Error	E'→ ε	E'→ ε
T	E → FT'	Error	Error	T'→ FT'	Error	Error
T'	Error	T'→ ε	T'→ *FT'	Error	T'→ ε	T'→ ε
F	F → id	Error	Error	F → (E)	Error	Error

Now the input string **id + id * id $** can be parsed using above table. At the initial configuration the stack will contain start symbol E, in the input buffer input string is placed

Stack	Input	Action
$E	id + id * id $	

Now symbol E is at top of the stack and input pointer is at first id ,hence M[E,id] is referred. This entry tells us E → TE', so we will push E' first then T.

Stack	Input	Action
$E'T	id + id * id $	E → TE'
$E'T'F	id + id * id $	E → FT'
$E'T'id	id + id * id $	F → id
$E'T'	+ id * id $	
$E'	+ id * id $	T' → ε
$E'T+	+ id * id $	E' → +TE'
$E'T	id * id $	
$E'T'F	id * id $	T → FT'
$E'T'id	id * id $	F → id
$E'T'	* id $	
$E'T'F*	* id $	T → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T → ε
$	$	E' → ε

Thus it is observed that the input is scanned from left to right and we always follow left most derivation while parsing the input string. Also at a time only one input symbol is referred to taking the parsing action. Hence the name of this parser is LL(1).The LL(1) Parser is a table driven predictive parser. The left recursion and ambiguous grammar is not allowed for LL(1) parser.

3.6 Bottom Up Parser –

In bottom up parsing method, the input string is taken first and we try to reduce this string with the help of grammar and try to obtain the start symbol. The process of parsing halts successfully as soon as we reach to start symbol.

The parse tree is constructed from bottom to up that is from leaves to root. In this process, the input symbols are placed at the leaf nodes after successful parsing.The bottom up parse tree is created starting from leaves, the leaf nodes together are reduced further to internal nodes, these internal nodes are further reduced and eventually a root node is obtained. The internal nodes are created from the list of Terminal and non terminal symbols. This involves -

Non Terminal for internal node = Non terminal ⋃ terminal

In this process, basically parser is tries to identify RHS of production rule and replace it by corresponding LHS. This activity is called reduction. Thus the crucial but prime task in bottom up parsing is to find the productions that can be used for reduction. The bottom up parse tree construction process indicates that the tracing of derivations are to be done in reverse order.

For example

Consider the grammar for declarative statement ,

$S \rightarrow TL;$

$T \rightarrow int \mid float$

$L \rightarrow L,id \mid id$

The input string is float id,id,id;

Step 1 : We will start from leaf node

Parse Tree

Step 1 :

Fig. i

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

"Handle of right sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right sentential form in rightmost derivation of γ"

For example

Consider the grammar

$$E \rightarrow E+E$$

$$E \rightarrow id$$

Now consider the string id+id+id and the rightmost derivation is

$$E \underset{rm}{\Rightarrow} \mathbf{E + E}$$

$$E \underset{rm}{\Rightarrow} E + \mathbf{E + E}$$

$$E \underset{rm}{\Rightarrow} E + E + \mathbf{id}$$

$$E \underset{rm}{\Rightarrow} E + \mathbf{id} + id$$

$$E \underset{rm}{\Rightarrow} \mathbf{id} + id + id$$

The underlined strings are called handles.

Right sentential Form	Handle	Production
id + id + id	id	E → id
E + **id** + id	id	E → id
E + E + id	id	E → id
E +E + E	E + E	E → E + E
E+E	E+E	E → E + E
E		

Thus bottom parser is essentially a process of detecting handles and using them in reduction.

3.6.1 Shift Reduce Parser –

Shift reduce parser attempts to construct parse tree from leaves to root. Thus it works on the same principal of bottom up parser. A shift reduce parser requires following data structures –

1. The input buffer storing the input string.
2. A stack for storing and accessing the LHS and RHS of rules.

The initial configuration of Shift reduce parser is as shown below

The parser performs following basic operations

Fig. 3.17 Initial Configuraiton

1. **Shift :** Moving of the symbols from input buffer onto the stack, this action is called shift.

2. **Reduce :** If the handle appears on the top of the stack then reduction of it by appropriate rule is done. That means RHS of rule is popped of and LHS is pushed in. This action is called Reduce action.

3. **Accept :** If the stack contains start symbol only and input buffer is empty at the same time then that action is called accept. When accept state is obtained in the process of parsing then it means a successful parsing is done.

4. **Error** A situation in which parser can not either shift or reduce the symbols, it can not even perform the accept action is called as error.

Let us take some examples to learn the shift-reduce parser

Example 3.3 : *Consider the grammar*

$E \rightarrow E\text{-}E$

$E \rightarrow E*E$

$E \rightarrow id$

*Perform Shift-Reduce parsing of the input string "id1-id2*id3"*

Solution :

Stack	Input Buffer	Parsing Action
$	id1-id2*id3$	Shift
$id1	-id2*id3$	Reduce by E→ id
$E	-id2*id3$	Shift
$E-	id2*id3$	Shift
$E-id2	*id3$	Reduce by E→ id
$E-E	*id3$	Shift
$E-E*	id3$	Shift
$E-E*id3	$	Reduce by E→ id
$E-E*E	$	Reduce by E→ E*E
$E-E	$	Reduce by E→ E-E
$E	$	Accept

Here we have followed two rules

1. If the incoming operator has more priority than in stack operator then perform shift.
2. If in stack operator has same or less priority than the priority of incoming operator then perform reduce.

➠ **Example 3.4 :** *Consider the following grammar –*

$S \rightarrow TL;$

$T \rightarrow int \mid float$

$L \rightarrow L,id \mid id$

Parse the input string int id,id; using shift reduce parser.

Solution :

Stack	Input Buffer	Parsing Action
$	int id,id;$	Shift
$int	id,id;$	Reduce by T→ int
$T	id,id;$	Shift
$Tid	id;$	Reduce by L→ id
$TL	id;$	Shift
$TL,	id;$	Shift
$TL,id	;$	Reduce by L → L, id
$TL	;$	Shift
$TL;	$	Reduce by S→ TL;
$S	$	Accept

3.6.2 Operator Precedence Parser –

A grammar G is said to be operator precedence if it posses following properties –

1. No production on the right side is ε
2. There should not be any production rule possessing two adjacent non terminals at the right hand side.

Consider the grammar for arithmetic expressions

E → EAE | (E) | -E | id

A→ + | - | * | / | ^

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

goto operation –

The function goto can be defines as follows –

If there is a production $A \rightarrow \alpha \bullet B\beta$ then goto($A \rightarrow \alpha \bullet B\beta$,B) = $A \rightarrow \alpha B \bullet \beta$ .That means simply shifting of • one position ahead over the grammar symbol(may be terminal or non terminal). The rule $A \rightarrow \alpha \bullet B\beta$ is in I then the same goto function can be written as goto(I,B).

Example 3.5 : *Consider the grammar*

$X \rightarrow Xb \mid a$

Compute closure(I) and goto(I).

Solution: Let

$I : X \rightarrow \bullet Xb$

Closure(I) = $X \rightarrow \bullet Xb$

= $X \rightarrow \bullet a$

The goto function can be computed as

goto(I,X) = $X \rightarrow X \bullet b$

$X \rightarrow X \bullet b$

Similarly goto (I, a) gives $X \rightarrow a \bullet$

Construction of canonical collection of set of item –

1. For the grammar G initially add $S' \rightarrow \bullet S$ in the set of item C.
2. For each set of items I_i in C and for each grammar symbol X(may be terminal or non terminal) add closure(I_i,X).This process should be repeated by applying goto(I_i,X) for each X in I_i such that goto(I_i,X) is not empty and not in C. The set of items has to constructed until no more set of items can be added to C.

Now we will consider one grammar and construct the set of items by applying closure and goto functions.

Example :

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

In this grammar we will add the augmented grammar E' → • E in the I then we have to apply closure(I)

I_0 :

E′ → • E

E → • E+T

E → • T

T → • T*F

T → • F

F → • (E)

F → • id

The item I_0 is constructed starting from the grammar E′→ • E. Now immediately right to • is E. Hence we have applied closure(I_0) and thereby we add E-productions with • at the left end of the rule.That means we have added E→ • E+T and E→ •T in I_0 .But again as we can see that the rule E→ • T which we have added, contains non terminal T immediately right to •. So we have to add T-productions in I_0. T→ •T*F and T→ • F.

In T-productions after • comes T and F respectively. But since we have already added T productions so we will not add those. But we will add all the F-productions having dots. The F→ • (E) and F→ • id will be added. Now we can see that after dot (and id are coming in these two productions. The (and id are terminal symbols and are not deriving any rule. Hence our closure function terminates over here. Since there is no rule further we will stop creating I_0.

Now apply goto(I_0,E)

E'→ • E
E→ • E + T
—Shift dot to right→
E'→ E •
E→ E • + T

Thus I_1 becomes

goto(I_0, E)

I_1: E′→ E •

E → E • +T

Since in I_1 there is no non terminal after dot we can not apply closure(I_1)

By applying goto on T of I_0

goto(I_0, T)

I_2: E → T •

T → T • *F

Since in I_2 there is no non terminal after dot we can not apply closure(I_2)

By applying goto on F of I_0

goto(I_0, T)

I_3: T→ F •

Since after dot in I_3 there is noting,hence we can not apply closure(I_3).

By applying goto on (of I_0. But after dot E comes hence we will apply closure on E,then on T,then on F.

goto(I_0,( )

I_4:T→ (• E)

E → • E+T

E → • T

T → • T*F

T→ • F

F→ • (E)

F→ • id

By applying goto on id of I_0

goto(I_0,id)

I_5: F → id •

Since in I_5 there is no non terminal to the right of dot we can not apply closure function here. Thus we have completed applying gotos on I_0. We will consider I_1 for applying goto. In I_2 there are two productions E′→ E • and E → E • + T.There is no point applying goto on E′→ E •, hence we will consider E′→ E • +T for application of goto.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We can design a DFA for above set of items as follows -

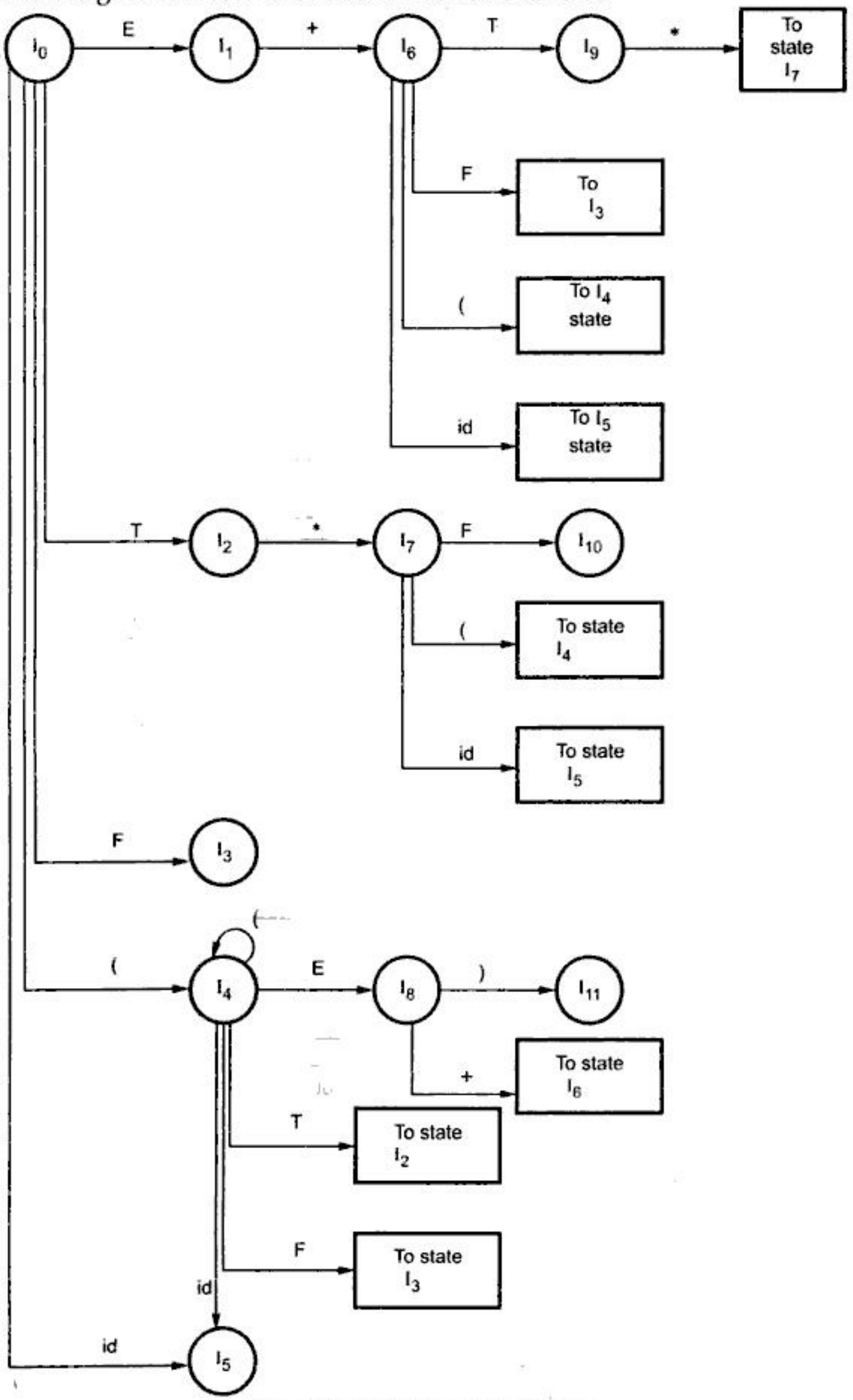
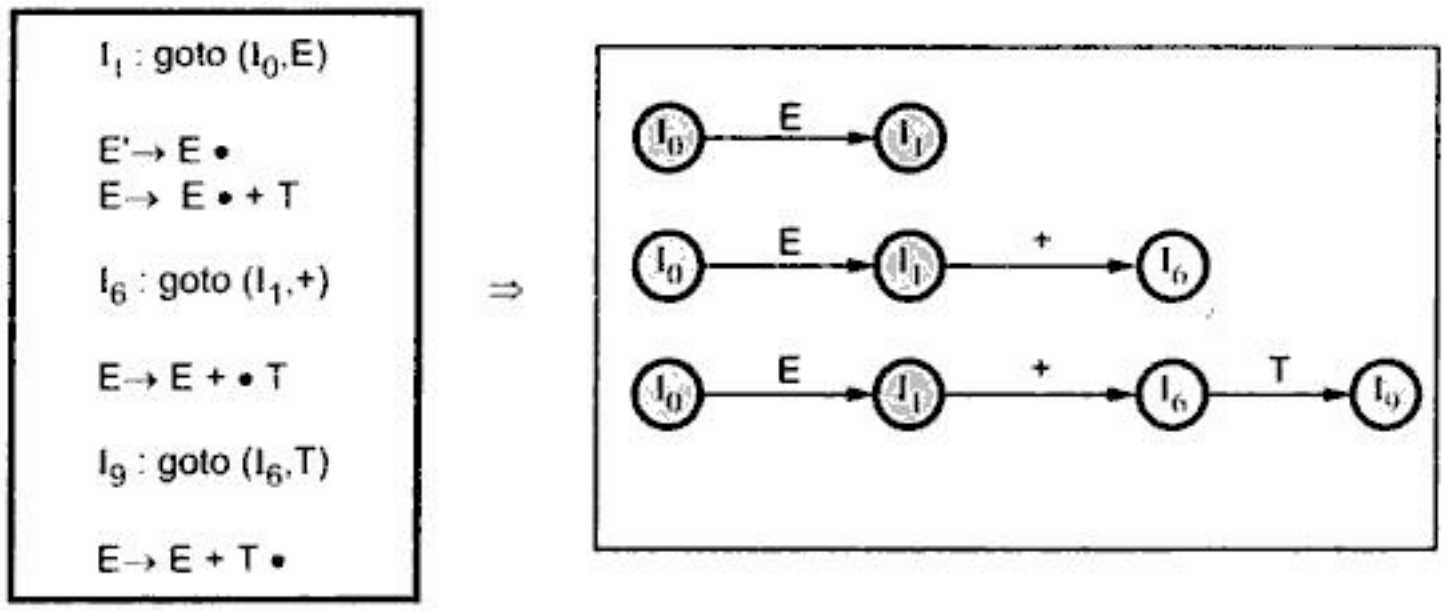


Fig. 3.22 DFA for set of items

In the given DFA every state is final state. The state I_0 is initial state. Note that the DFA recognizes viable prefixes.

For example - For the item



The viable prefixes E, E+ and E+T are recognized here continuing in this fashion The DFA can be constructed for set of items. Thus DFA helps in recognizing valid viable prefixes that can appear on the top of the stack.

Now we will also prepare a FOLLOW set for all the non terminal as we need FOLLOW according to rule 2.b of parsing table algorithm.

[Note for readers: Below is a manual method of obtaining FOLLOW. We have already discussed the method of obtaining FOLLOW by using rules.This manual method can be used for getting FOLLOW quickly]

FOLLOW(E') =As E' is a start symbol $ will be placed

={$}

FOLLOW(E) :

E'→ E that means E'=E =start symbol. ∴ we will add $.

E → E+T the + is following E ∴ we will add +.

F → (E) the) is following E ∴we will add)

∴ FOLLOW(E) = {+,),$}

FOLLOW(T) :

As E' → E, E → T the E'=E=T=start symbol. ∴we will add $.

E → E+T

E → T+ T ∵ E → T

The + is following T hence we will add +.

$T \rightarrow T * F$

As * is following T we will add *

$F \rightarrow (E)$

$F \rightarrow (T)$ $\because E \rightarrow T$

As) is following T we will add)

$\therefore$ FOLLOW(T) ={+,*,),$}

FOLLOW(F) :

As $E' \rightarrow E$, $E \rightarrow T$ and $T \rightarrow F$ the E'=E=T=F=start symbol. We will add $.

$E \rightarrow E+T$

$E \rightarrow T+T$ $\because E \rightarrow T$

$E \rightarrow F+T$ $\because T \rightarrow F$

The + is following F hence we will add +.

$T \rightarrow T * F$

$T \rightarrow F * T$ $T \rightarrow F$

As * is following F we will add *

$F \rightarrow (E)$

$F \rightarrow (T)$ $E \rightarrow T$

$F \rightarrow (F)$ $T \rightarrow F$

As) is following F we will add)

FOLLOW(F) ={+,*,),$}

Building of parsing table

Now From canonical collection of set of items Consider I_0

I_0:

$E' \rightarrow \bullet E$

$E \rightarrow \bullet E+T$

$E \rightarrow \bullet T$

$T \rightarrow \bullet T*F$

$T \rightarrow \bullet F$

$F \rightarrow \bullet (E)$

$F \rightarrow \bullet id$

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

FOLLOW(E)={+,),$}

Hence add rule E → T in the row of state 2 and in the column of +,) and $. In the given example of grammar E → T is rule number 2. ∴ action[2,+] = r2, action[2,)]=r2,action[2,$]=r2.

Similarly, Now in state I_3

T→ F •

A →α • rule 2b

A=T, α= F

FOLLOW(F)={+,*,),$}

Hence add rule T→ F in the row of state 3 and in the column of +,*,) and $. In the given example of grammar T →F is rule number 4.Therefore action[3,+]=r4, action[3,)]=r4, action[3,$]=r4.Thus we can find the match for the rule A→α • from remaining states I_4 to I_{11} and fill up the action table by respective "reduce" rule. The table with all the action[ ] entries will be

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4					
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4					
5		r6	r6		r6	r6			
6	S5			S4					
7	S5			S4					
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Now we will fill up the goto table for all the non terminals. In state I_0, goto(I_0, E)=I_1 Hence goto[0,E]=1,similarly goto(I_0, T)=I_2. Hence goto[0,T]=2.Continuing in this fashion we can fill up the goto entries of SLR parsing table.

Finally the SLR(1) parsing table will look as –

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Remaining blank entries in the table are considered as syntactical errors.

Parsing the Input using parsing table –

Now it's the time to parse the actual string using above parsing table. Consider the parsing algorithm

Input: The input string w that is to be parsed and parsing table.

Output: Parse w if $w \in L(G)$ using bottom up. If $w \notin L(G)$ then report syntactical error.

Algorithm:

1. Initially push 0 as initial state onto the stack and place the input string with $ as end marker on the input tape.
2. If s is the state on the top of the stack and a is the symbol from input buffer pointed by a lookahead pointer then

a) If action[s,a]=shift j then push a,then push j onto the stack.Advance the input lookahead pointer.

b) If action[s,a]=reduce $A \rightarrow \beta$ then pop $2^*|\beta|$ symbols.If i is on the top of the stack then push A,then push goto[i,A] on the top of the stack.

c) If action[s,a]=accept then halt the parsing process. It indicates the successful parsing.

Let us take one valid string for the grammar

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T^*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

Input string : id*id+id

We will consider two data structures while taking the parsing actions and those are – stack and input buffer

Stack	Input buffer	action table	goto table	Parsing action
$0	id*id+id$	[0,id]=s5		Shift
$0id5	*id+id$	[5,*]=r6	[0,F]=3	Reduce by F → id
$0F3	*id+id$	[3,*]=r4	[0,T]=2	Reduce by T → F
$0T2	*id+id$	[2,*]=s7		Shift
$0T2*7	id+id$	[7,id]=s5		Shift
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	Reduce by F → id
$0 T2*7F10	+id$	[10,+]=r3	[0,T]=2	Reduce by T → T*F
$0T2	+id$	[2,+]=r2	[0,E]=1	Reduce by E → T
$0E1	+id$	[1,+]=s6		Shift
$0E1+6	id$	[6,id]=s5		Shift
$0E1+6id5	$	[5,$]=r6	[6,F]=3	Reduce by F → id
$0E1+6F3	$	[3,$]=r4	[6,T]=9	Reduce by T → F
$0E1+6T9	$	[9,$]=r1	[0,E]=1	E → E+T
$0E1	$	accept		Accept

In the above table at first row we get action[0,id]=s5,that means shift id from input buffer onto the stack and then push the state number 5. On the second row we get action[5,*] as r6 that means reduce by rule 6,F→ id, hence in the stack id is replaced by F. By referring goto[0,F] we get state number 3 hence 3 is pushed onto the stack.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

I_8:goto(I_3,C)
$C \rightarrow aC \bullet$, a/d

For I_4 and I_5 there is no point in applying gotos. Applying gotos on a and d of I_6 gives I_6 and I_7 respectively. Hence we will apply goto on I_6 for C for the rule.

$C \rightarrow a \bullet C$, \$

I_9:goto(I_6,C)
$C \rightarrow aC \bullet$, \$

For the remaining states I_7,I_8 and I_9 we can not apply goto.Hence the process of construction of set of LR(1) items is completed.Thus the set of LR(1) items consists of I_0 to I_9 states.

Construction of canonical LR parsing Table –

To construct the Canonical LR Parsing table first of all we will see the actual algorithm and then we will learn how to apply that algorithm on some example. The parsing table is similar to SLR parsing table comprised of action and goto parts.

Input : n Augmented grammar G′.

Output : The canonical LR parsing table.

Algorithm:

1. Initially construct set of items $C=\{I_0,I_1,I_2...I_n\}$ where C is a collection of set of LR(1) items for the input grammar G′.
2. The parsing actions are based on each item I_i. The actions are as given below-

a) If [$A \rightarrow \alpha \bullet a \beta$, b] is in I_i and goto(I_i, a)= I_j then create an entry in the action table action[I,a]=shift j.

b) If there is a production [$A \rightarrow \alpha \bullet$, a] in I_i then in the action table action[I_i,a] = reduce by $A \rightarrow \alpha$. Here A should not be S′.

c) If there is a production $S' \rightarrow S\bullet$, \$ in I_i then action[i,\$] = accept.

3. The goto part of the LR table can be filled as :The goto transitions for state i is considered for non terminals only. If goto(I_i,A)=I_j then goto[I_i,A]=j.
4. All the entries not defined by rule 2 and 3 are considered to be "error".

Example 3.8 : *Construct the LR(1) parsing table for the following grammar –*

(1) $S \rightarrow CC$

(2) $C \rightarrow aC$

(3) $C \rightarrow d$

Solution: First we will construct the set of LR(1) items -

I_0:
S'→ • S,$
S→ • CC,$
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0,S)
S'→ S •,$

I_2: goto (I_0,C)
S→ C • C,$
C→ • aC,$
C→ • d,$

I_3: goto (I_0,a)
C→ a • C, a/d
C→ • aC, a/d
C→ • d, a/d

I_4: goto (I_0,d)
C→ d • , a/d

I_5: goto (I_2,C)
S→ CC •, $

I_6:goto (I_2,a)
C'→ a• C,$
C→ • aC,$
C→ • d,$

I_7:goto (I_2,d)
C→ d•, $

I_8:goto (I_3,C)
C→ aC•, a/d

I_9:goto (I_6,C)
C→ aC•, $

The DFA for the set of items can be drawn as follows.

See Fig. 3.23 on next page

Now consider I_0 in which there is a rule matching with $[A \rightarrow \alpha \bullet a\beta, b]$ as

C→ • aC, a/d and if the goto is applied on a then we get the state I_3. Hence we will create entry action[0,a]=shift 3. Similarly,

In I_0

$C \rightarrow \bullet d \quad a/d$

$A \rightarrow \alpha \bullet a\ \beta,\ b$

$A=C,\ \alpha = \varepsilon,\ a=d,\ \beta = \varepsilon,\ b=a/d$

$goto(I_0,d)=I_4$

hence action[0,d]=shift 4

For state I_4

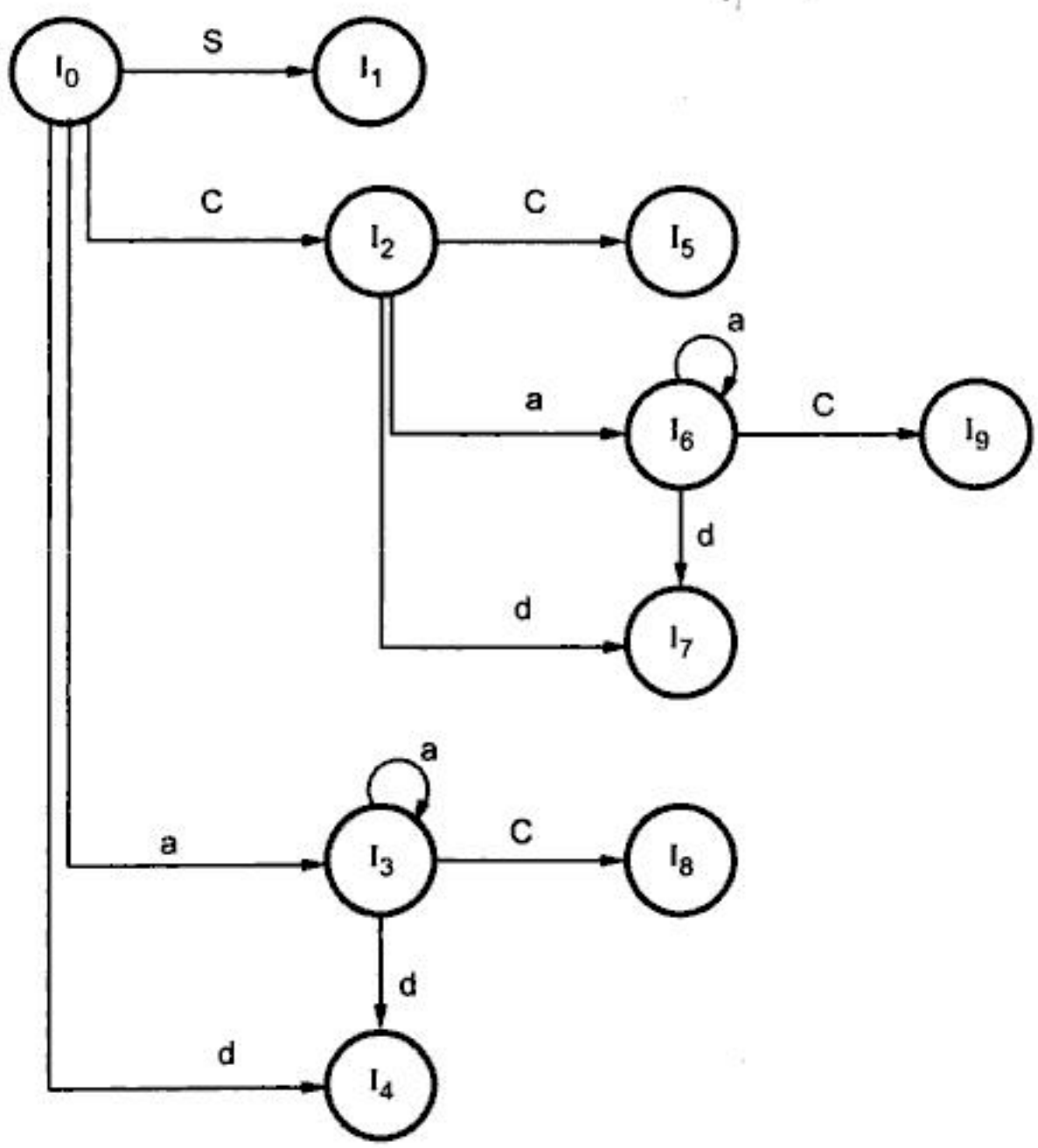


Fig. 3.23 DFA [goto graph]

$C \rightarrow d \bullet, a/d$

$A \rightarrow \alpha \bullet, a$

$A=C, \alpha = d, a=a/d$

action[4,a] = reduce by $C \rightarrow d$ i.e. rule 3

$S' \rightarrow S\bullet$, \$ in I_1

So we will create action[1,$]=accept.

The goto table can be filled by using the goto functions.

For instance goto(I_0,S)=I_1.Hence goto[0,S]=1. Continuing in this fashion we can fill up the LR(1) parsing table as follows -

	action			goto	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

The remaining blank entries in the table are considered as syntactical error.

Parsing the Input using LR(1) parsing table –

Using above parsing table we can parse the input string"aadd" as

Stack	Input buffer	action table	goto table	Parsing action
$0	aadd$	action[0,a]=s3		
$0a3	add$	action[3,a]=s3		Shift
$0a3a3	dd$	action[3,d]=s4		Shift
$0a3a3d4	d$	action[4,d]=r3	[3,C]=8	Reduce by C→ d
$0a3a3C8	d$	action[8,d]=r2	[3,C]=8	Reduce by C→ aC
$0a3C8	d$	action[8,d]=r2	[0,C]=2	Reduce by C→ aC
$0C2	d$	action[2,d]=s7		Shift
$0C2d7	$	action[7,$]=r3	[2,C]=5	Reduce by C→ d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the given input string is successfully parsed using LR parser or canonical LR parser.

3.7.3 LALR Parser

In this type of parser the lookahead symbol is generated for each set of item. The table obtained by this method are smaller in size than LR(k) parser. In fact the states of SLR and LALR parsing are always the same. Most of the programming languages use LALR parsers.

We follow the same steps as discussed in SLR and canonical LR parsing techniques and those are

1. Construction of canonical set of items along with the lookahead.
2. Building LALR Parsing table
3. Parsing the input string using canonical LR Parsing table

Construction set of LR(1) items along with the lookahead.

The construction LR(1) items is same as discussed in section 3.7.2. But the only difference is that: in construction of LR(1) items for LR parser, we have differed the two states if the second component is different but in this case we will merge the two states by merging of first and second components from both the states.

For example in section 3.7.2 we have got I_3 and I_6 because of different second components, but for LALR parser we will consider these two states as same by merging these states.i.e.

$I_3 + I_6 = I_{36}$

Hence

I_{36} :goto(I_0,a)

$C \rightarrow a \bullet C$, a/d/$

$C \rightarrow \bullet aC$, a/d/$

$C \rightarrow \bullet d$, a/d/$

Let us take one example to understand the construction of LR(1) items for LALR parser.

Example 3.8 :

$S \rightarrow CC$

$C \rightarrow aC$

$C \rightarrow d$

Construct set of LR(1) items for LALR parser

I_0:
S'→ • S,$
S→ • CC,$
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0,S)
S'→ S •,$

I_2: goto (I_0,C)
S→ C • C,$
C→ • aC,$
C→ • d,$

I_{36}: goto (I_0,a)
C→ a • C, a/d/$
C→ • aC, a/d/$
C→ • d, a/d/$

I_{47}: goto (I_0,d)
C→ d • , a/d/$

I_5: goto (I_2,C)
S→ CC •, $

I_{89}:goto (I_3,C)
C→ aC • a/d/$

We have merged two states I_3 and I_6 and made the second component as a or d or $. The production rule will remain as it is. Similarly in I_4 and I_7 .The set of items consist of states { I_0, I_1, I_2, I_{36}, I_{47}, I_5, I_{89}}.

Construction of LALR Parsing Table –

The algorithm for construction of LALR parsing table is as given below –

1. Construct the LR(1) set of items.
2. Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij}=I_i$ U I_j.
3. The parsing actions are based on each item I_i.The actions are as given below-

a) If [A →α • a β, b] is in I_i and goto(I_i, a)= I_j then create an entry in the action table action[I,a]=shift j.

b) If there is a production [A→α • , a] in I_i then in the action table action[I_i,a] = reduce by A→α. Here A should not be S'.

c) If there is a production S'→ S •, $ in I_i then action[i,$] = accept.

4. The goto part of the LR table can be filled as :The goto transitions for state i is considered for non terminals only. If goto(I_i,A)=I_j then goto[I_i,A]=j.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

$0a36a36C89	d$	action[89,d]=r2	[36,C]=89	Reduce by C→ aC
$0a36C89	d$	action[89,d]=r2	[0,C]=2	Reduce by C→ aC
$0C2	d$	action[2,d]=s47		Shift
$0C2d47	$	action[47,$]=r36	[2,C]=5	Reduce by C→ d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the LALR and LR parser will mimic one another on the same input.

3.8 Comparison of LR Parsers –

It's the time to compare SLR, LALR and LR parser for the common factors such as size, class of CFG, efficiency and cost in terms of time and space.

1. SLR parser is smallest in size. It is easiest method of LR parsers. This method is based on FOLLOW function.
2. The LALR and SLR have the same size.
3. Canonical LR parser or LR parser is largest in size.
4. LALR is applicable to a wider class of context free grammar than that of SLR.
5. LR parsers are most powerful then LALR and then SLR in the LR family. But most of the syntactic features can be expressed in the grammar of LALR.
6. LALR reacts differently than LR on erroneous inputs. Error detection is immediate in LR but not immediate in LALR.
7. The time and space complexity in LALR and LR is more. But efficient methods exist for constructing LALR parsers directly.

3.9 Using Ambiguous Grammar –

As we have seen various parsing methods in which if at all the grammar is ambiguous then it creates the conflicts and we can not parse the input string with such ambiguous grammar. But for some languages in which arithmetic expressions are given ambiguous grammar are most compact and provide more natural specification as compared to equivalent unambiguous grammar. Secondly using ambiguous grammar we can add any new productions for special constructs.

While using ambiguous grammar for parsing the input string we will use all the disambiguating rules so that each time only one parse tree will be generated for that specific input. Thus ambiguous grammar can be used in controlled manner for parsing the input. We will consider some ambiguous grammar and let us try to parse some input string.

Using precedence and associativity to resolve parsing action conflicts

Consider an ambiguous grammar for arithmetic expression –

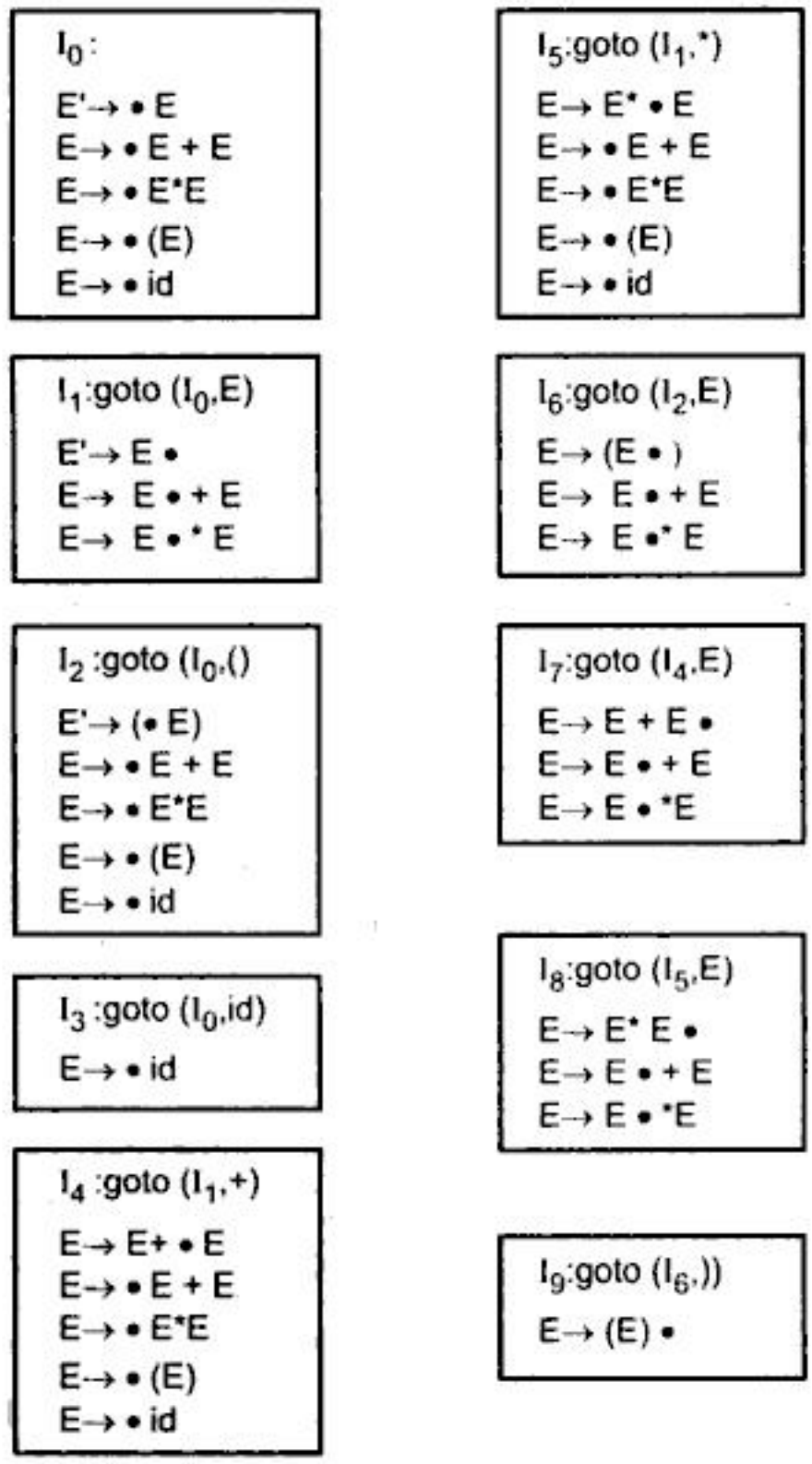
$E \rightarrow E+E$

$E \rightarrow E^*E$

$E \rightarrow (E)$

$E \rightarrow id$

Now we will build the set of LR(0) items for this grammar.



Here FOLLOW(E)={+,*,),$}.

We have computed FOLLOW (E) as it will be required while processing.

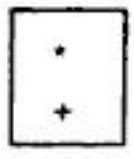
Now using the same rules of building SLR(0) parsing table we will generate the parsing table for above set of items

State	action						goto
	id	+	*	(	)	$	E
0	S3			S2			1
1		S4	S5			Accept	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		S4 or r1	S5 or r1		r1	r1	
8		S4 or r2	S5 or r2		r2	r2	
9			r3		r3	r3	

As we can see in the parsing table the shift/reduce conflicts occure at state 7 and 8. We will try to resolve it,how? Let us consider one string **"id+id*id"**.

Stack	Input	Action with conflict resolution
$0	id+id*id$	Shift
$0id3	+id*id$	Reduce by E → id
$0E1	+id*id$	Shift
$0E1+4	id*id$	Shift
$0 E1+4id3	*id$	Reduce by E→ id
$0E1+4E7	*id$	The conflict can be resolved by shift 5 ←
$0 E1+4E7*5	id$	Shift
$0 E1+4E7*5id3	$	Reduce by E → id
$0 E1+4E7*5E8	$	Reduce by E → E*E
$0E1+4E7	$	Reduce by E → E+E
$0E1	$	Accept

As * has precedence over + we have to perform multiplication operation first. And for that it is necessary to push * on the top of the stack.The stack position will be



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Technical Publications Pune
1, Amit Residency, 412 Shaniwar Peth, Pune - 411030
☎(020) 24495496/97, Email : technical@vtubooks.com
Visit us at : www.vtubooks.com

Rs. 230/-

ISBN 81-8431-064-1